2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PRIVATE AND SHARED DATA IN OBJECT-ORIENTED PROGRAMMING

by

Vassilios Theologitis

March, 1992

Thesis Advisor: Michael L. Nelson

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School — 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PRIVATE AND SHARED DATA IN OBJECT-ORIENTED PROGRAMMING

12. PERSONAL AUTHOR(S)
Vassilios Theologitis

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 01/90 TO 03/92	1992, March, 19	176

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Object-Oriented, shared data,concurrency, distributed systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In a typical object-oriented system, there are two kind of variables: those which are private to instances(objects) and those which are shared by all instances of a class. Variables may also be declared in some object-oriented languages as private, public, or subtype visible which affects the acess to the data. However we know of no object-oriented programming which allows data(variables) to be daclared as private for specific methods only. The purpose of this thesis is to propose a solution to the problems of implementing and maintaining both shared and private data at various levels within an object-oriented environment..

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael L. Nelson	22b. TELEPHONE (Include Area Code) (408) 646-2026 — 22c. OFFICE SYMBOL CS/Ne

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PRIVATE AND SHARED DATA IN OBJECT-ORIENTED PROGRAMMING

by
Vassilios Theologitis
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

Author: [illegible]
Vassilios Theologitis

Approved By: [illegible]
Michael L. Nelson, Thesis Advisor

[illegible]
Gary J. Hughes, Second Reader

[illegible]
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

In a typical object-oriented system, there are two kind of variables: those which are private to instances(objects) and those which are shared by all instances of a class. Variables may also be declared in some object-oriented languages as private, public, or subtype visible which affects the access to the data. However we know of no object-oriented programming which allows data(variables) to be declared as private for specific methods only. The purpose of this thesis is to propose a solution to the problems of implementing and maintaining both shared and private data at various levels within an object-oriented environment.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Completing this thesis, I would like to express my sincere appreciation and gratitude to my advisor, Dr. Michael L. Nelson, for his assistance and guidance.

I dedicate this thesis to my wife, Mimi, for her great help, support, and encouragement during all this period, including late nights and weekends that were spent in writing this thesis. Finally a special thanks is due to the Hellenic Navy for giving me this opportunity to study at the Naval Postgraduate School.

I. INTRODUCTION

A. BACKGROUND

Object-oriented programming (OOP) is a new approach to programming. Many people believe that OOP is the future of programming languages and also that it can improve the development of software. The main reason for this evolution of programming languages is the necessity of the human mind to be able to express ideas, and also to be able to more easily model real world activities with a computer program. Ever since the first computer languages appeared, the human mind has always tried to find ways to express thoughts and ideas more easily. This is the main reason why man builds newer high level languages every day which are closer to his way of thinking.

Thus OOP is rapidly becoming a popular approach to the construction of complex software systems. Benefits of object-orientation include support for modular design, code sharing, reuse, and extensibility.

The main purpose of this research is to observe different types of variables in object-oriented programming in different kinds of environments. The initial starting point for this research stems from the problems encountered during the development of an object-oriented model of the software controller of the Naval Postgraduate School (NPS) Autonomous Underwater Vehicle (AUV) [BN91]. The

C. ORGANIZATION

The remainder of this thesis is divided into four chapters. Chapter II introduces the basic concepts of object-oriented programming, concurrency, and distributed systems. Chapter III investigates variables in an object-oriented programming environment with a special attention to sharing data. In Chapter IV, various solutions are suggested. Chapter V contains the conclusions and recommendations for future research. The computer code developed during the course of this thesis is contained in Appendices A through G.

II. BACKGROUND

In this chapter we survey the literature on object-oriented concepts, concurrency, and distributed systems.

A. OBJECT-ORIENTED CONCEPTS

1. Basic Concepts

In this section we give a brief description of the basic concepts of object-oriented programming. Since this area is still relatively young, there are no standard definitions yet within the object-oriented community [Nel90a]. However the following basic concepts are found in most object-oriented languages.

a. Classes

"Objects which share the same behavior are said to belong to the same class" ([WWW90]: pp.22). A class defines a group of similar objects with the same structure and behavior. Any object generated from a given class has the same set of information (the structure) and abilities (the behaviors). The structure of a class is represented by the variables, and the behavior is represented by the methods existing for each class.

A brief (but good) description of a class is given in the following statement:

"Class is indeed the technical term that will be applied in object-oriented languages to describe such sets of data structures characterized by common properties." ([Mey88]: pp.52)

We can picture a class as a factory that produces products with the same *main properties*, with no limits to the number of products. We cannot have two objects with different structures within the same class. For example, consider a class car; it can be thought of as vehicle factory that produces a certain car model. All objects of the class have the same structure.

Classes can be related to one another by inheritance; we say that a subclass inherits the structure and behaviors (i.e., the variables and methods) of its superclass. The subclass may, in turn, serve as the superclass for another subclass. This leads to a hierarchy of classes in which we can talk about ancestor and descendant classes. This will be discussed further in Section 2.

b. Objects

An object is an instance of a class. Objects that belong to the same class have common structures and behaviors. Actually, we can say that the object is the first and main element in object-oriented programming because when we start thinking about how to define the class we first try to specify and categorize each needed object. Several objects from the same class can exist at the same time, and all of them have the same *main properties*. As we will see later, these are the variables and methods.

Referring to our previous example, the class **car** can be used to produce several instances, each with the same basic structure and set of behaviors. **Our_car**, a particular car with our options (color, whether it is automatic, etc.), is an instance of the class **car**. Since an object is generated from a class, the variables and the methods of the class exist for every object. That is, **our_car** will have the same set of variables and methods that every other instance of the class **car** has.

c. *Methods*

The operations or procedures that an object knows how to perform are called methods. A method is similar to a function definition or procedure call in conventional languages. A method deals only with objects of the class within which it is contained (defined), and can be activated only by sending a message to the object. A message consists of the name of a method along with any required arguments including the name of the object and any parameters. Each time an object receives a message it performs the requested operation by executing the appropriate method.

Another definition of method is that it is the step by step algorithm executed in response to the received message where the name in the message matches the name of the method [WWW90]. Sending a message is more general than calling a function because different objects can respond to the same message in different ways; as we will see later, this is known as polymorphism.

In our car example, we might have the method **start_engine** defined for the **car** class. We could then send this message to **our_car** in order to start its engine.

Methods can be categorized as either class methods or instance methods. Messages sent to a class cause a class method to be executed while messages sent to objects cause an instance method to be executed.

d. Variables

There are two kinds of variables in an object-oriented language: class variables and instance variables.

(1) Class Variables. "A class variable is shared both in name and value by all instances of a class" ([Nel91a]: pp.4). We can consider these variables to be *global* for any object of that particular class.

For example, in our **car** class we could have the class variable **number_of_wheels** which is the same for every instance (object) of this class. This variable is the same for every object. Therefore, if it were to change, all objects immediately change as they all share access to this same class variable.

(2) Instance Variables. "An instance variable is shared in name only, not in value, by all instances of a class" ([Nel91a]: pp.4). Whereas class variables can be thought of as *global* to all the instances of a class, instance variables can be considered as the private data of each object.

In our example **car** class we could have the instance variable **serial_number**. All car objects will have this variable, but the value for each object is different.

e. Private, Public, and Subtype Visibility

Both variables and methods can be divided into three categories which define their visibility to subclasses and end-users. These three categories are as follows:

- Public
- Private
- Subtype visible

When a variable or method is declared to be public we mean that anyone has access to it from inside or outside the class. A public variable may have the additional attributes of read-only or writable outside the owner class (i.e., anyone may be able to read it, but they may not be able to change it). When we declare a method to be public we also mean that it is a part of the published or public interface. We must point out here that since public variables and methods can be accessed by anyone, it can be very costly to change them [RB91].

The private property is the opposite of public. When we declare data or methods to be private we mean that they can only be reached by methods declared within that class. Private methods and variables are *internal* to a class.

Therefore we can modify them or even delete them more easily as this will only impact other methods of the same class.

Subtype-visible is something between these two extremes. Subtype-visible variables or methods can be reached only by methods declared inside the same class or its descendant classes. End-users, however, cannot directly access these variables or methods.

2. Properties of Object-Oriented Languages

a. Reusability

"Reusability is the ability of a system to be reused, in whole or in part in order to construct a new system" ([Mic88]: pp.13). Reusability is one of the major advantages of object-oriented languages because it reduces the cost of designing, coding, and testing.

(1) Instantiation. This term is used in object-oriented languages when we generate an instance (object) of a class. When we say instatiate we mean that we create a new object of a class. Every time we instatiate an object we are reusing the class definition. That is, each time that we instatiate an object a class serves as a template which provides the variables and methods.

In object-oriented programming we can have statically or dynamically instantiated objects. Statically instantiated objects must be declared/instantiated at compile time, whereas dynamically instantiated objects may be instantiated at run time.

(2) Inheritance. "Inheritance can be defined simply as a code sharing mechanism" ([Nel91a]: pp.5). An inherited class may be defined as an extension or restriction of another class [Mey88]. In other words, inheritance allows us to reuse the definition of a previous class in the creation of a new one. This new class is called a subclass of the first one, which is called the superclass. In the case that we can inherit from only a single class we have a simple or single inheritance. If, on the other hand, we can inherit from many classes we have multiple inheritance (MI).

What about inherited class variables in the subclass? There are two different aspects in the way that class variables can be implemented in inheritance. The first notion implements the class variable as a global variable for all classes related by inheritance, which implies the ability to change the value from any class. That is, a single class variable is shared by all the classes related by inheritance. Alternatively, changing a value in any class does not modify the value of the other classes related by inheritance. That is, the class variable is in effect duplicated for each new subclass. [Nel90a]

(3) Polymorphism. "Polymorphism (sometimes called operator overloading or function overloading) can be defined as allowing different data types (classes) to have methods (routines) with the same name which may be implemented differently" ([Nel90a]: pp.4). The ability of different objects to respond to the same message is called as polymorphism. Even when the same

message is sent from the same place in code, it can invoke different methods, depending on the object it is sent to [SB86].

There are two forms of polymorphism: simple polymorphism and multiple polymorphism [Nel90a, Mic88]. With simple polymorphism, each class may have its own implementation of an operation. With multiple polymorphism, a single class may have several operations with the same name.

(4) Genericity. "Genericity is the ability to parameterize modules. The need for this facility is particularly clear for classes representing general data structures: arrays, lists, trees, matrices etc" ([Mey88]: pp.104). One common form of genericity is the abstract data type (ADT). An ADT is a data structure with a set of associated operators in which the implementation details are hidden, allowing the user to reference the ADT with implementation-independent code. This allows the physical implementation of the ADT to be changed without affecting the user-written code. Since the class definition is a form of an ADT, the class itself represents one form of genericity. [Nel91a]

The generic module is a module pattern and is not directly usable. Instances of the generic module are obtained by providing real types for each of the generic parameters. It is a technique that is used to avoid some of the requirements of static type checking.

b. Extensibility

Extensibility is the facility in a software system to change or modify anything we need in accordance with our requirements in order to produce a new class from an existing one. This is achieved through the use of inheritance in object-oriented languages.

Extensibility is easy when dealing with small and simple programs, but is more difficult in larger programs. Thus, the problem of extensibility is a problem of scale because as the programs grow larger, the problem of adaptation also becomes harder [Mey88].

There are two principles essential to improve extensibility in the designing of classes [Mey88]:

- Design simplicity: it is easier to adapt changes when we have a simple structure in a class rather than a complex one.
- Decentralization: we have more possibilities that the changes will be in one class and will not affect a chain reaction of changes over the whole system.

3. Object-Oriented Languages

Object-oriented languages can be categorized into two families. The first family contains those languages with the object-oriented features added to an existing language, and we refer to this family as *bolted-on* languages. The second family contains those languages which are designed and constructed around the principles of object-oriented programming, and we refer to this family as *built-in* languages.[Nel90a]

a. Classic-Ada

Classic-Ada [Sof89, NM92] is an object-oriented preprocessor for Ada [Boo87]. It is a preprocessor because it converts the programs written in the Classic-Ada language into standard Ada, and the resulting programs are then compiled with a standard Ada compiler. It adds to Ada the concepts and features of an object-oriented language, including methods, objects, classes, and inheritance (thus, Classic-Ada is a *bolted-on* object-oriented language).

Class definitions are divided into specifications and bodies, similar to package specifications and bodies in Ada. The terminology and reserved words used in Classic-Ada are very similar to the basic concepts of other object-oriented languages. Classic-Ada supports single inheritance but does not support multiple inheritance. Every class needs at least one method, the method *create* for the creation of objects from this class; this method is a class method. If we want to have a method by which objects of classes are reclaimed we need a second method *delete,* which is an instance method. These are the only methods that we cannot inherit from an ancestor class; they must be declared anew for each new class.

Methods and variables are classified by the reserved word *instance* if they are instance methods or instance variables. If they are class methods or variables we do not put any reserved word before them as the default is a class method or variable[1].

Classic-Ada does not support the concepts of private, public, and subtype visible. All variables are by default subtype-visible, and all methods are public.

To accomplish dynamic binding Classic-Ada uses the reserved word *send*, which is a fairly common approach for many *bolted-on* object-oriented languages [NM92].

Since Classic-Ada is an extension of the Ada Language, which does support concurrency, a concurrent Classic-Ada program is therefore possible [Nel91b].

b. *C++*

C++ [WP88] is an extension of the popular C language, adding special features for object-oriented programming. Thus, C++ is a *bolted-on* language. C++ was originally a preprocessor for the C language, but C++ compilers are now available.

C++ supports encapsulation, combining data abstraction with methods to manipulate data into a class-type object. The reserved words *class*, *union*, and *struct* are used for the declaration of a class. The major difference

[1]The Classic User's Manual [Sof89] does not specifically mention class variables and class methods. This 'feature' was discovered through experimentation.

between them is the accessibility of the members, because many versions of C++ language support the concepts of public, private, and protected (note that in C++ terminology, the term protected is used rather than subtype-visible). All versions of C++ support single inheritance, and multiple inheritance is now supported in some versions.

Polymorphism in C++ is accomplished by placing the reserved word *virtual* before the functions (methods). Virtual functions allow you to use many versions of the same function throughout a class hierarchy, with the particular version to be executed being determined at run time.

Messages are sent to objects using a mechanism similar to that used to invoke a function (object_name.function _name(argument)). Since C is a pointer language we can also send a message with a pointer if we have a pointer that points to an object (object_pointer -> function_name (argument)).

C++ provides a special type of member function with the reserved word *constructor*. A *constructor* specifies how a new object of a class type will be created. The deallocation of the memory is achieved with the method *destructor*.

C++ does not include any constructs for handling concurrency.

c. *Smalltalk*

One of the first true object-oriented languages was Smalltalk [Seb89]. According to our previously discussed classification of object-oriented languages, Smalltalk is a *built-in* language.

Smalltalk is an interpretive language which uses an intermediate compiler. As an interpretive language it provides rapid testing of incremental changes to the image. Many problems can be solved by using or modifying existing classes and methods.

All programming in Smalltalk is accomplished by sending messages to objects. It supports three kinds of messages: unary, binary, and keyword. Smalltalk supports both class and instance methods.

Smalltalk variables come in two varieties. The first is *private* which means they are local to an object, and the second is *shared* which means they are visible outside the object in which they are declared. Determination in the program is achieved by using lower or uppercase letters. Private variables must begin with lowercase letters, while shared variables begin with uppercase letters.

Inheritance in Smalltalk is also supported. However, only a single hierarchy is supported, and all classes must be descendants of the root class **object**. We have the ability to rename, modify, or add any inherited variable or method.

Smalltalk includes a *Fork*[2] construct, so we can run processes concurrently. But we must point out that they are essentially run sequentially on

[2]Fork, is a control structure for indicating parallelism. It creates two concurrent processes, one at label and one at statement following the Fork statement.[Dei90]

a uniprocessor. The reason for this is that each process runs until finished or stopped for a reason and then the next process starts to run, etc. A *Yield* statement which is also provided by Smalltalk allows processes to yield to the processor at any time, but true non-determinism is not possible in standard Smalltalk. [SN90]

d. Actor

Actor[3] is very similar to Smalltalk, and Actor is also a ***built-in*** object-oriented language. Actor is an interpretive language which provides rapid testing of incremental changes to the image. Many applications can be built by using or modifying existing classes and methods.

Inheritance is also supported in Actor. Like Smalltalk, only single inheritance is supported, and all classes must be descendants of the root class **object**. We have the ability to rename, modify, or add any inherited variable or method.

Actor supports both class and instance methods and also class and instance variables. Class variables must start with the sign '$' and then follow with the first letter being capitalized, instance variables must not capitalize the first letter and there is no special character in front. Although Actor has global variables that are similar to those in traditional languages, they are accessible

[3]It should be realized that we are discussing the language Actor which is registered trademark of the Whitewater Group, Inc. (a registered servicemark) [Act90]. This language is not associated with the Actor model [Agh88].

from anywhere in the program. They are used most often during the testing of the program.

Actor does not include any constructs for handling concurrency.

B. CONCURRENT PROGRAMMING

1. Conventional Concurrency

Concurrent execution is conventionally viewed in terms of autonomous sequential processes executing in parallel [BLW87]. In concurrency we usually deal with processes which may run at the same time. These processes may run separately from one another, which means that there is no exchange of information between the processes. Each one starts and stops processing without waiting on the others for any reason.

It is also possible that we need some level of cooperation between the processes. This case is called asynchronous, which means that the processes may require occasional synchronization and cooperation [Dei90].

Thus the correct behavior of a concurrent program is dependent upon the necessary synchronization and communication between its processes. Synchronization is concerned with the action(s) that have to occur in one process before an action of another process. Communication deals with the information passing between processes.

The problems we need to solve in a concurrent programming environment include the following:

a. *Message Passing*

Message passing is simply a way of passing information from one process to another. Message passing between two processes involves four issues [BLW87]. These four issues are:

- process naming
- synchronization
- message structure
- failure on communication

Process naming deals with the address that a message is sent to. We can directly address the process that is going to receive a message. Alternatively, we can just name the channel or the communication port where the message is to be sent, and it is assumed that the receiver will eventually receive the message. The first case has the advantage of simplicity, and the disadvantage that we cannot change the receiver of the message at run-time. The second case has the advantages that we can change the message destination at the run-time, or we can have an anonymous process as a receiver.

Synchronization is the need for the mutual acknowledgement between the sender and receiver of a message. When the sender continues executing immediately after sending the message we have asynchronous message passing. When the sender waits (i.e., is blocked) until the receiver accepts the message we have synchronous message passing.

Virtually any data structure can be transmitted within a message, possibly subject to size limitations imposed by the system.

The necessity of handling a communication failure is more of a necessity in networks and distributing systems.

b. Shared Variables

Shared variables in concurrent programs are the case in which we have data shared between different processes. When one process accesses the shared variable, we must keep any other process wishing to do the same from doing so until the first process finishes access to the shared variable[4]. We say that each process requires exclusive access, which is referred to as mutual exclusion. When a process is accessing shared data it is said to be in a critical section, and the shared data is often called critical data. [Dei90]

2. Concurrency in Object-Oriented Programming

In the real world many things exist and do things concurrently. Object-oriented programming systems show great potential for use in designing and building concurrent systems as many objects can exist and do things concurrently. [Nel90c]

In concurrent object-oriented programming there are many ways that we can have concurrency. Different objects could execute various methods at the

[4] Access to shared variables may also be modeled after the Readers and Writers problem [Dei90]. This approach allows any number of readers (with no active writer) or a single writer (with no active readers) at any point in time.

same time, different methods can be executing in the same object at the same time, or a single object (instance) could be executing a single method which does several things concurrently. Of course, these three possibilities can be combined - that is, several objects, each executing several methods, each of which does several things concurrently. [Nel90c]

Unfortunately, we still have most of the same problems that we have with any concurrent language when we move into the object-oriented environment. The major problem is in controlling the concurrent activities during program execution.

Concurrency can be achieved in a computer system which has only a single processor (i.e., a uniprocessor)[5] or in a computer system which contains two or more processors (i.e., a multiprocessor). In both cases we have to deal with the messages that objects send to one another to achieve successful execution.

C. DISTRIBUTED PROGRAMMING

1. Conventional Distributed Systems

A distributed computer system contains multiple autonomous processing elements cooperating for a common purpose or to achieve a common goal. Distributed systems can be divided into two main categories [BT88]:

[5] Obviously, we can only have simulated concurrency on a uniprocessor as only one thing may execute at a time.

- tightly coupled: systems that have a common memory;
- loosely coupled: systems that do not have a common memory.

Note that communication between processes in a loosely coupled system must be via some form of message passing. Thus, these two categories can also be thought of as those which communicate through shared data, and those which communication via message passing.

One efficient implementation for shared data in distributed system is the use of replication [BT88]. Another possibility is the use of the same techniques that are used in concurrent systems, such as mutual exclusion, monitoring, etc.

The most important differences between shared data and message passing are as follows [BT88]:

1. The process sending a message must know some form of identity of the receiver.
2. When making a new assignment there is a delay in sending the message from one process to another; with shared data the assignment has an immediate effect.
3. A message sent from one process to another is more secure than shared data which is reachable by anyone.
4. To exchange messages between processes we may need synchronization. However to access shared data we definitely need synchronization between the processes (i.e., this is a classical mutual exclusion problem).
5. When passing a message it maybe difficult to pass a complex data structure; with shared data this difficulty does not exist.

2. Distributed Object-Oriented Programming

The main goal in distributed object-oriented programming systems is to establish a distributed object manager that allows several different systems to share objects [BT88].

The development of distributed systems has been partly motivated by the desire to extend the limited set of sharable resources of a particular computer system to a large, possibly unlimited, set of network resources. Resource sharing poses the problems of naming, protection, and consistency.

There are currently several development projects concentrating on high performance system kernels for distributed systems. They all support a client-server approach, in which the servers can manage data on behalf of external clients. An alternative approach is to have direct invocation of (possibly remote) objects. Each object is executed in its own virtual address space. [Hor90]

III. STUDYING VARIABLES IN OBJECT-ORIENTED SYSTEMS

In this chapter we analyze and define the sharing of variables in an object-oriented environment. As discussed in Chapter II, a class consists of variables which are defined in the class definition. The values of these variables have one of the following properties: they are either shared between the instances (objects) of a class or they are independent for each instance (object) of a class; that is, they are either class variables or instance variables.

Before observing the variables and sharing of data in an object-oriented system, it is first necessary to define the execution environment. We begin with a brief description of the different terms to be used. Unfortunately, these terms are often used in a rather confusing manner within the computer science community - the same term may be used differently or different terms may be used the same [Nel90c].

A *process* is the smallest part of a program which may be controlled as a separate entity. A *concurrent process* is simply one which may be executed at the same time as other concurrent processes. *Multiprocessing* is the ability to give the appearance of executing two or more processes concurrently, regardless of the number of processors involved. A *parallel* system is one which supports true multiprocessing (i.e., two or more processors are used).

A *uniprocessor* is a computer system which has only a single processor. A *multiprocessor* is one which contains two or more processors.

In this chapter we discuss the various types of execution in an object-oriented programming environment and the results obtained with different types of variables. We will classify the execution type as either sequential or concurrent.

Figure 1[6] shows how class definitions can be presented in a language-independent manner. In this example, the class Alpha has one class variable (cv), one instance variable (iv), and three methods (Method_X, Method_Y, and Method_Z). The class Beta defines no new variables or methods, but inherits all of the variables and methods defined for the class Alpha.

Class Alpha	
Superclass:	none
Class variable:	cv
Instance variable:	iv
Methods:	Method_X
	Method_Y
	Method_Z
Class Beta	
Superclass:	Alpha
Class variable:	none
Instance variable:	none
Methods:	none

Figure 1 Class Alpha and Beta Definitions

[6]The language -independent class definitions are modeled after those found in [Nel90a].

We can define several instances (objects) for each class. For example, objectA1, objectA2, and objectA3, for class Alpha; and objectB1, objectB2, and objectB3 for class Beta.

Figure 2 gives one representation of the classes and their instances. The classes are presented in this way as it is similar to block scoping diagrams used for conventional language systems, and it makes the concept of sharing data in an object-oriented environment more understandable. Note, however, that this diagram does not show that a single class variable (cv) is shared by both Alpha and Beta.

Several Classic-Ada programs have been generated to study various aspects of variables. Various tests are conducted in both sequential and concurrent modes of execution.

A. SEQUENTIAL EXECUTION

The term sequential execution is used in those cases where processes are executed sequentially. Although sequential execution is the simplest case, the effects of inheritance cannot be omitted. Thus, both classes with inheritance and classes without inheritance are studied.

1. Classes Without Inheritance Relation

Classes without inheritance are simply ones which are not descended from other classes. Thus, we will only consider a class Alpha.

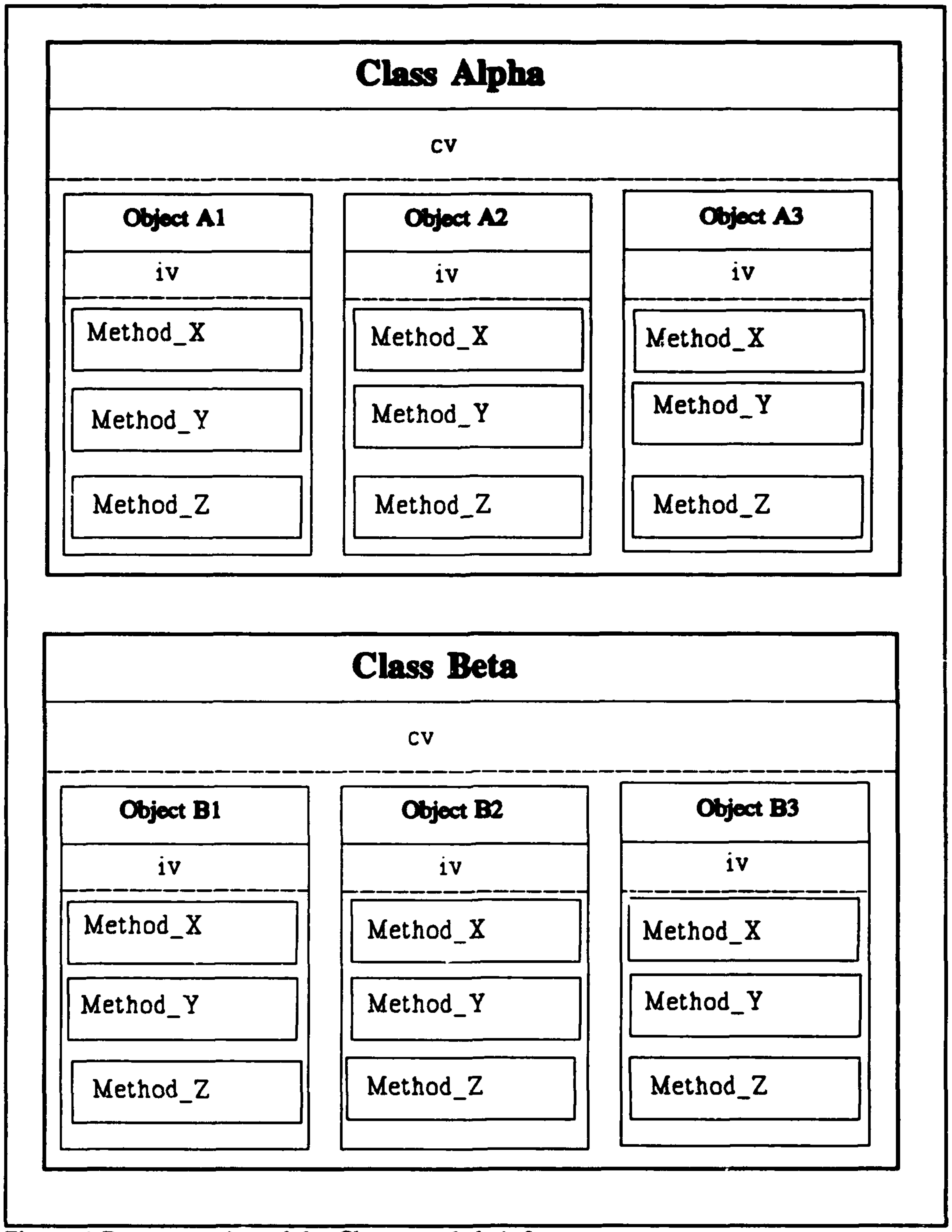

Figure 2 Representation of the Classes and their Instances

a. Class Variable

As discussed in Chapter II, a class variable is a variable shared by name and value in both all instances (objects) of the class. To illustrate the use of class variables in a sequential environment, we will define the class Alpha as presented in Figure 3.

Class Alpha

Superclass:	none
Class variable:	cv1
Instance variable:	none
Methods:	Set_class_variable
	Get_class_variable
	Create
	Delete

Figure 3 Class Alpha Definition

The method Set_class_variable sets the value of the class variable (cv1), while the Get_class_variable returns its current value. The Create and Delete methods are used to create and delete instances (objects) of the class.

The files giving the specification and implementation of the class Alpha are *Alpha_spec_CV.ca* and *Alpha_body_CV.ca*.[7] The main program, which creates instances of the class Alpha and manipulates them, is in the file

[7] In Classic-Ada the declaration of a class requires two files. The first is the specification file which defines the name of the class and the names and types of its methods. The second is the body of the class where we specify the implementation of the methods and also define the variables and their types.

program_CV_ one.ca and its output is in the file *program_CV_one.script*. All files discussed in this section are contained in Appendix A, Section A.

This experiment was used to check the value of the class variable for different instances (objects) in the same class. This was done to ensure that the class variable was indeed shared in both name and value between different instances of a class. We create several instances (objects) of the class Alpha and then apply the methods in the following order.

First, we declare the object and create it as an instance (object) of class Alpha using the Create method.

Second, we invoke the Get_class_variable method to get the current value of the class variable for the instance. As can be seen in the output of the program, the first time that we call the Get_class_variable method for the first object the value is null since no value has been defined yet. For subsequent instances (objects), the value is that given to the previously defined instance (object).

Third, we invoke the Set_class_variable to set the class variable to a specific value. For our program this can be any legal character.

Fourth, we again invoke the Get_class_variable to get the current value of the class variable at this instance. As can be seen in the output, the value of the class variable is equal to the character given in the Set_class_variable method, as expected.

Finally, we invoke the Delete method to delete the instance.

Looking at the output, we can see that Classic-Ada implements class variables as expected - they are shared in both name and value by all instances of a class. Note that we delete each instance before creating the next. These deletions have no effect on the class variable as it exists within the class Alpha. Its value is maintained even after the instance that last set the value is deleted.

As a further test, we run another program (*program_ CV_two.ca*) with all the Delete messages for the instances of the class at the end of the program. The output of this new main procedure is in the file *program_CV_ two.script*. In this way we keep the instances of the class Alpha 'alive' until the end of the main procedure. As expected, however, this coexistence of all instances does not change the previous results. It also shows that all instances see the latest value of the class variable.

b. Instance Variables

An instance variable is shared in name only by all instances of a class, as discussed in the last chapter. We now define the class Alpha as shown in Figure 4 to test this feature. The files with the specification and implementation of class Alpha are the *Alpha_spec_IV.ca* and *Alpha_body_IV.ca*. All the files used in this section are included in Appendix A, Section B.

The main program file is *program_IV.ca*. Our goal here is to observe the value responses of an instance variable through different instances of a class. The output of the main procedure of the program is presented in the file

Class Alpha

Superclass:	none
Class variable:	iv1
Instance variable:	none
Methods:	Set_instance_variable
	Get_instance_variable
	Create
	Delete

Figure 4 Class Alpha Definition

program_IV.script. In this program we create several instances of class Alpha and apply methods in the following order.

First, we create an object and then invoke the Get_instance_variable method to get the current value of the instance variable for this instance. As excepted, this returns a null value as objects are created without any initial or default value for this variable.

Next, we invoke the Set_instance_variable to set the instance variable with a value. Finally, we invoke the Get_instance_variable method to get the current value of the instance variable, which is the value set by the previous Set_ instance_variable method, as expected.

As can be seen in the output of this program, the values of the instance variable of each instance are independent of all other instances of that class. Indeed, instance variables are shared in name but not in value by all the objects of a class.

2. Classes With Inheritance Relation

Remembering that inheritance is a *sharing mechanism*, we expect to have some sharing of data between classes. Since we have code reuse in inheritance, the reuse of a class implies that we share the names and the types of the existing variables, and the names and parameters of the existing methods.

a. Class Variables

We will now use the classes Alpha and Beta as shown in Figure 5. Notice that in Classic-Ada it is necessary to define the methods Create and Delete for each class, even though it is appears that the methods should have been inherited from superclass [NM92]. The class Beta inherits all other variables and methods defined in the class Alpha.

Class Alpha	
Superclass:	none
Class variable:	cv1
Methods:	Set_class_variable
	Get_class_variable
	Create
	Delete
Class Beta	
Superclass:	Alpha
Class variable:	none
Instance variable:	none
Methods:	Create
	Delete

Figure 5 Class Alpha and Beta Definitions

The files of class Alpha and Beta are *Alpha_spec_CV.ca* and *Alpha_body_CV.ca*, and the *Beta_spec_CV.ca* and *Beta_body_CV.ca* respectively. All files used in this section are contained in Appendix B, Section A.

In this experiment we want to observe the response of class variable thought the inheritance mechanism between different objects.The main program is *program_CV_inher.ca,* and its output is contained in *program_CV_inher.script*. Objects are created and manipulated in the following order:

- First instance of class Alpha (object A1)
- First instance of class Beta (object B1)
- Second instance of class Alpha (object A2)
- Second instance of class Beta (object B2)

We then invoke messages in the following order.

First, we declare and create the objects.

Second, we invoke the Get_class_variable to get its current value. As expected, the first time that we call the Get_class_variable method on the first object the current value of class variable is null since we do not define any default or initial value. After that, we get the value set by the previous object, also as expected.

Third, we invoke the Set_class_variable to set the class variable to a new value.

And finally, we again invoke the Get_class_variable method. As expected, the value is that which was set in the previous Set_class_variable method.

Notice that the value set for each instance is reflected as the 'initial' value for the next object, regardless of class membership (i.e., Alpha or Beta). Thus, the class variable is shared in both name and value between all instances of all classes related by inheritance.

b. *Instance Variables*

For this experiment we use the classes Alpha and Beta as shown in Figure 6.

Class Alpha	
Superclass:	none
Class variable:	iv1
Methods:	Set_instance_variable
	Get_instance_variable
	Create
	Delete
Class Beta	
Superclass:	Alpha
Class variable:	none
Instance variable:	none
Methods:	Create
	Delete

Figure 6 Class Alpha and Beta Definitions

The files with the specification and implementation of class Alpha and Beta are the *Alpha_spec_IV.ca* and *Alpha_body_IV.ca* and *Beta_spec_IV.ca* and *Beta_body_IV.ca*, respectively. The main program is the *program_IV_inher.ca*, and its output is in the file *program_IV_inher.script*. All files for this section are in Appendix B, Section B. Objects are created and manipulated in the following order:

- First instance of class Alpha (object A1)
- First instance of class Beta (object B1)
- Second instance of class Alpha (object A2)
- Second instance of class Beta (object B2)

Methods are invoked on the objects in the following order:

First, objects are declared and created.

Second, we invoke the Get_instance_variable method to get the initial value of the instance variable for this instance. As can be seen in the output of this program, this value is null as we did not define any initial or default value for the instance variable.

Third, we invoke the Set_instance_variable method to set the instance variable with a given value. For our program this could be any legal character.

Fourth, we invoke the Get_instance_variable method to get the current value of the instance variable for this instance. As expected, the value returned is that set by the previous Set_instance_variable method.

Finally, after following these four steps for each instance, we again send each instance the message Get_instance_variable. As expected, each object has maintained its own value for the instance variable.

3. Sequential Execution Summary

The class variable, as pointed in Chapter II, depends on how the object-oriented language being used supports inheritance. One of the following properties are supported:

- it is shared by all the classes related by inheritance.
- it is duplicated for each new subclass, so changing the value in one class does not modify the other classes.

However, the class variable in each class is always a shared variable for that particular class. This is as we observed in the previous experiments. This means that every instance of the class has access to this variable, and nothing outside the class can access it without using the appropriate method. Classic-Ada, however, supports the first notion above as an inherited class variable is shared by all instances of each class related by inheritance. It is not possible to have any kind of shared variable between two classes if there is no inheritance relation[8].

Each instance of a class has its own private set of instance variables. In other words, memory storage is allocated to maintain the internal representation for each instance of a class. Thus, the instance variable is a shared variable, but only within an object; more specifically, it is shared between the methods of the object. The problem arises if we want shared data between only some methods, but not between others. Once again consider Figure 2. For example, what if we would like to have a variable that only Method_X in object A1 can access (i.e., Method_Y and Method_Z cannot access it)? Alternatively, how could we have a variable shared by Method_X and Method_Y in an instance (object) such as A1, but not by a Method_Z? This is not possible in any object-oriented programming language that we know of. This is one concept that we are investigating in this thesis. Our proposals will be discussed in Chapter IV.

B. CONCURRENT EXECUTION

With the term concurrent execution we mean the ability to have two or more processes running at the same time; that is, some form of multiprocessing. This could be a parallel system with several processors (i.e., a multiprocessor), or a

[8] It is obvious that we can use some form of a global variable in the program, but this is not acceptable since the majority of literature on the designing object-oriented system software advises readers to avoid using this kind of variable.

uniprocessor with simulated multiprocessing (via some form of context switching).

1. Concurrency In A Uniprocessor Environment

It should be realized that in this type of concurrency we do not have any real concurrent execution, as everything is actually being executed sequentially.

a. *Class Variable*

For this experiment we use the same class Alpha as previously defined and shown in Figure 3. We want to observe the responses of the class variable in a simulated concurrent environment.

The main program, which creates and manipulates instances of the class Alpha, is in the file *program_CV_conc.ca.* Its output is in the file *program_CV _conc.script.* All files used in this section are in Appendix C, Section A.

We have four tasks in the main program. Three are responsible for creating and manipulating objects, and the fourth task is the main procedure where we just print a simple message. To make sure that all the tasks are eligible to start running at the same time we give the same priority to each of the task, (pragma priority (1)). After creating an object, the three tasks apply methods in the following order:

- Get_class_variable to get the current value of the class variable for the object.

- Set_class_variable to set the class variable with a new value.
- Get_class_variable again to get the current value of the class variable for the object.

The main task just prints the word "main" so that we can observe when this task runs. We expect after the beginning of the main program that the three tasks run in parallel with the main one. Note that before the main procedure can end the other tasks in the system have to finish. The output of the execution program indicates that the tasks executed in the following order:

- main procedure begins
- object A3 created and manipulated
- object A2 created and manipulated
- object A1 created and manipulated
- main procedure ends

The reason that the system chose to run the task manipulating object A3 before the other tasks is purely arbitrary as the same priority was specified for all. The results also indicate that the tasks did finish before the main procedure ended.

As expected, the value of class variable for each instance of for each task is reflected as the 'initial' value for the next object, regardless of which task created and manipulated the object. Thus, the class variable is shared between all objects regardless of which task is involved.

Notice that in the main program that we put the invocation of the methods delete at the end of the main procedure (task). This is because we knew the running order of the tasks (i.e., that the main task finishes after all others) from previous experiments with all tasks with the same priority. Any statement after the 'main' runs after the finishing of the other tasks.

b. Instance Variable

For this experiment we again use the class Alpha as shown in Figure 4. We want to illustrate the use of instance variables in a simulated concurrent environment.

The main program is the *program_IV_conc.ca,* and its output is in the file *program_IV_conc.script*. All the files for this section are in Appendix C, Section B. We have the same structure as the previous main program *(program_CV_conc.ca)* except that we now have an instance variable rather than a class variable.

The final output was as expected (in accordance with the class variable results), with the tasks executing in following order:

- main procedure begins
- object A3 created and manipulated
- object A2 created and manipulated
- object A1 created and manipulated
- main procedure ends

As expected, the value of instance variable is autonomous and individualistic for each object; that is, it was not shared between the objects of the class. Thus, an instance variable in a concurrent environment is still shared only in name and not by value between objects of the same class.

2. Concurrency in a Multiprocessor Environment

Parallel processing can be divided into two basic architectures: Shared Resource and Distributed Resource systems, as shown in Figure 7 [INM89]. Shared Resource systems execute the components of a problem on conventional CPU's. They are connected by a common bus to shared memory. A Distributed Resource system executes the parallel parts of a problem among hardware nodes. Each node runs its own program and includes a CPU with local memory. In our experiments, each node is a Transputer[9].

3. OOP In A Parallel System

In this section we study variables in a multiprocessor environment. All programs used in this section were developed in Classic-Ada and executed on a Transputer.

[9] Transputers are microprocessors built by Inmos, the English semiconductor manufacturer. A Transputer is a single VLSI device with processor, memory, and communications links for direct connection to other transputers. They operate as a stand alone machine, or as a node in a network interconnected via links. When in a network, each transputer operates on its own using only on chip memory and programs. Communication from one processor to another occurs over the links, each of which has a dedicated link interface. The communication interface is implemented in hardware and does not need the processor for its control. [INM89]

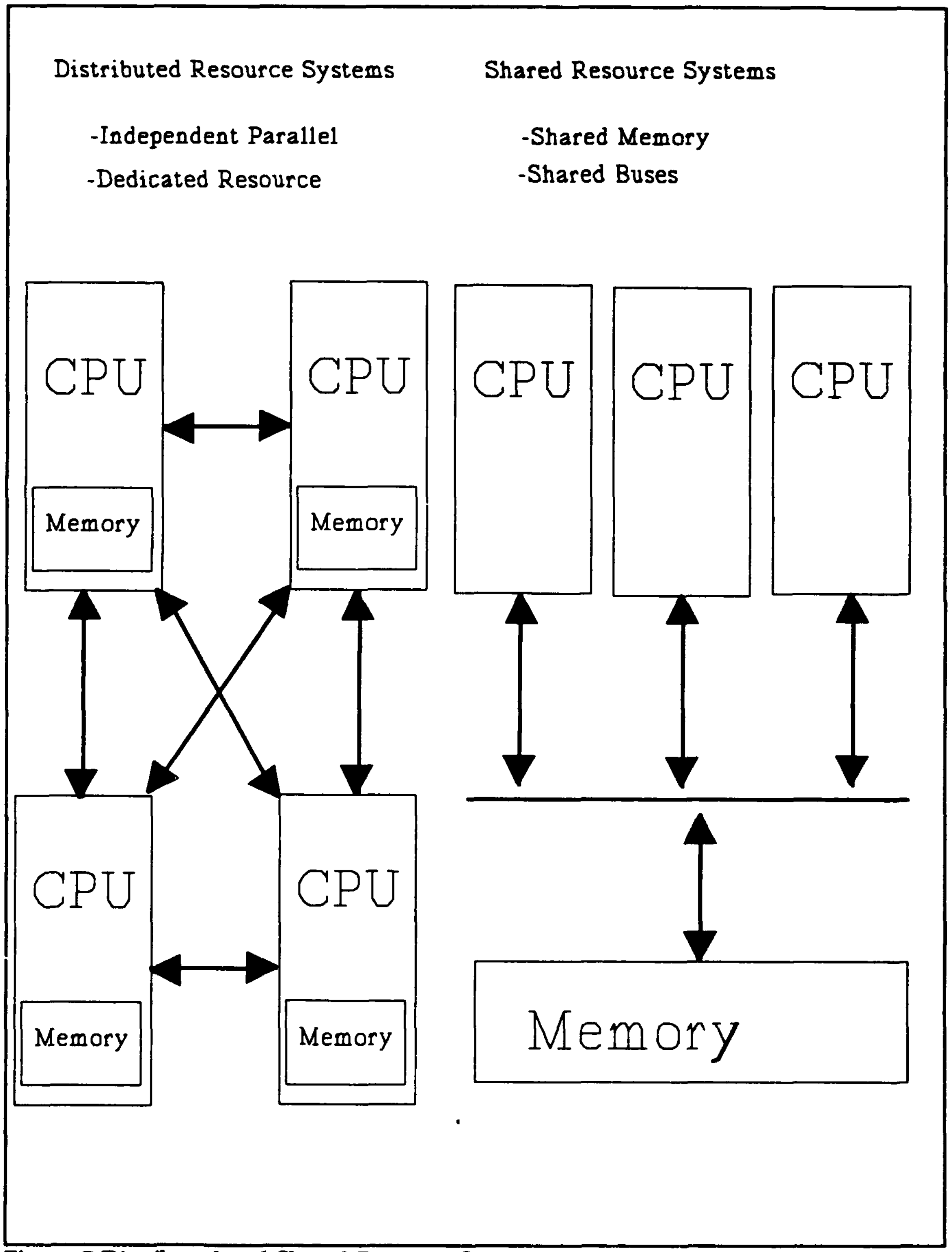

Figure 7 Distributed and Shared Resource Systems

Before continuing, however, it is first necessary to give a brief description of the environment.

a. Transputer

A Transputer is a single device with processor, memory, and communications links for direct connections to other Transputers.

(1) Communication. Link communications run simultaneously with processor computation to maximize the performance of distributed systems. Each link carries information bidirectionally on two wires between a pair of transputers in the computing network. The links provide for direct communication between processes on neighboring transputers. Communication across the link uses a link protocol and is accomplished as a sequence of single byte transmissions.

(2) Memory. Transputers are not designed to share memory; instead, each has its own dedicated memory. The transputer also has a small amount of on-chip memory for faster access. On the T800 this is four Kbytes of static RAM. Also, four Gbytes of addressable external memory is possible.

b. Two Objects, Each On Separate Transputers

For this experiment we run each instance on a separate transputer. Thus, we expect to observe the problems previously discussed about concurrent environments in distributed systems. For this experiment we use the class Alpha as shown in Figure 3. All the programs used in this section are contained in Appendix D. Since we have two transputers, we need to have one program for each of them. Thus, we created two programs named *program_CV_trans.ca* which are nearly identical; they differ only in the creation of different objects (one generates the object A1 and the other object A2). The important issue in each program was to create one instance (object) and then change the value of the class variable to see if that change would affect the object residing on the other transputer. We compiled each program separately, then bind it with each group of files for each transputer, and then run the program. These two almost identical files are the *Alpha_one.ada* and *Alpha_two.ada* files.

We must point out here that the type of the class variable has been changed from character to integer as we encountered difficulties in debugging the configuration for the occam[10] files that we were using for the channels. Using integers, it was easier to configure the channels, unfortunately, the manuals [AA90] were not much help in this problem.

Communication in the Classic-Ada programs was achieved via the implementation package CHANNELS. Both programs made use of the package COMMON which declared the data types used in channel communication. The

[10] The transputer architecture directly implements the process model of concurrency to describe parallel systems naturally and simply. This logical model is the basis of occam, the first general purpose language with built in support for both concurrency and communication. Occam is used to program transputers in a way that closely resembles real-world systems. It can also be easily combined with conventional high level languages in different ways to create parallel descriptions of problems. Occam and transputers were designed to complement each other in a powerful way. Occam is not only based on concepts of concurrency but also on communication concepts that relate directly to the transputer links.[INM89]

package COMMON also contained an instantiation of the generic package CHANNEL_IO which provided channel read and write operations. This package ensured that each program had a consistent view of the data communicated between them. Communication between the two programs occurred as follows:

- After the Set_class_variable method in the random.ada file we put the statement WRITE, so that the value of class_variable1 would be put on the channel; from then on any other transputer existing in the network could READ the value of this variable.
- After the first Get_class_variable method in the sieve.ada file we put the statement READ, so that we could get the new value of class_variable1 from the channel.

The output file *program_CV_trans.script* is also included in Appendix D. It is important to realize that since transputers do not have shared memory, each time the value changed it was necessary to send/receive to/from the channel in order to update the other objects. Thus, the user has to establish the point where the update of the values takes place. However, this means that the class variable does not always have the same value in all instances (objects) of a class.

It is shared only by all instances residing on a single transputer until it is explicitly passed from one transputer to another. Thus, the class variable can no longer be considered to the shared between all instances of the class.

4. Concurrent Execution Summary

As we observe in the sequential execution section, a class variable has the property of being shared between all objects of the same class and also between classes related by inheritance.

In our concurrent programs in a uniprocessor environment we have reinforced the idea that on a class variable shared between all objects, regardless of which task is involved. What we have actually done is run a multiprocessing program on a uniprocessor to simulate a shared resource system (i.e., shared memory), since the opportunity to work on a distributed system with common memory as presented in Figure 7 did not exist.

In a parallel system the experiment was conducted in a transputer environment (i.e., a distributed resource system) as depicted in Figure 7. The class variable has its value shared between all instances of a class on a single transputer, but not between transputers. Upon reflection this makes sense as what happened is that we essentially created several different (but duplicate) classes which just happen to have the same name. That is, the transputer is a distributed resource system and since the class definition is loaded (duplicated) on each processor's memory, then it should be expected that changing value of class variable on one processor should have no affect on its value in another processor.

IV. METHOD VARIABLES

As we have seen, variables in an object-oriented environment take one of two forms: they are either class variables or instance variables. Class variables belong to the class; that is, they are shared by all instances of the class. Instance variables belong to individual objects. Although there are a few variations, such as how the class variable is implemented when inherited and various options that modify the visibility of the variables, these are the only two kinds of variables available.

It is questionable whether these two types of variables are enough for all situations. It has been suggested that we may wish to implement methods as individual processes with their own private state of variables [BN91]. This is one possible solution to the problem of implementating/modeling software processes in an OO environment. We may have several processes, each with its own private set of data, but they all manipulate a common ('global') set of data - this is not possible in any object-oriented programming language that we know of.

Building on the results of the last chapter, we will now propose and experiment with various forms of 'method variables' in an attempt to expand the number and types of variables available in an object-oriented environment.

Each object has a protocol, which is the set of messages that it can respond to (this is also called the external interface of the object). It is simply the collection of methods defined for the instances of its class. Each time we send a message to an instance (object), the method corresponding to the message's selector is executed. Typically, methods have formal parameters, and the values of the message's argument bind the formal parameters of the method before executing the method's code. The state of an object can be retrieved and updated through its methods. However, the method has no state of its own. According to [KA90], methods can be categorized as:

- Those methods whose primary purpose is to retrieve or update variables.
- More general methods performing complex computations.

But our question is: can we have methods that perform complex computations using variables that are not accessible by other methods? That is, could a method maintain variables for itself only for a single instance (object)? We will now develop the general idea of an object-oriented language with this property. More specifically, we want to simulate the ability for a method to have data that is not accessible by other methods defined for the class. We begin by defining the concept of method variables, then explore how they are used in both sequential and concurrent execution environments.

A. METHOD VARIABLES

We now introduce the *method variable (mv)*, which can take the form of either a method class variable or a method instance variable, depending on which properties we want to give to the variable. The *method instance variable (miv)* is accessible by only a single method of a single object. The *method class variable (mcv)* is shared by all objects of a class, but is only accessible by a single method.

Method variables can easily be incorporated into the class definition[11] (see Figure 8). Each time we instantiate an object, it has the method variables that are defined in the class definition for the given methods. Our approach to method variables follows the general properties of variables in an object-oriented language. That is, we can divide method variables into two main categories:

- Method variables based on instance variables that are part of the private data of each object (method instance variables).
- Method variables based on class variables that are shared by every object in the class (method class variable).

Thus, each method variable inherits the properties of the type of variable where it is based on.

Figure 8 shows how class definitions with method variables can be presented in a language independent manner. We use the example of class Alpha from the last chapter, adding method variables to some of the methods.

[11]Since Classic-Ada class definitions are in two files, the specification and the body, the method variable will be declared in the body.

```
Class Alpha
  Superclass:          none
  Class variables:     cv
  Instance variables:  iv
  Methods :
    Method_X
      Method_instance_variables:   X_miv1, X_miv2
      Method_class_variables:      X_mcv
    Method_Y
      Method_instance_variables:   none
      Method_class_variables:      none
    Method_Z
      Method_instance_variables:   Z_miv
      Method_class_variables:      Z_mcv
```

Figure 8 The Class Definition with Method Variables

The class Alpha has one class variable (cv), one instance variable (iv), and three methods (Method_X, Method_Y, and Method_Z). Method_X has two method instance variables (X_miv1 and X_miv2) and a method class variable (X_mcv). Method_Y has no method class variables or method instance variables. Method_Z has one method instance variable (Z_miv) and one method class variable (Z_mcv).

1. Method Instance Variables

Method instance variables are based on instance variables. They can be thought of simply as instance variables that are only accessible by a single method. In Figure 8, method instance variables are declared along with the method itself. In our test applications, however, this is not possible. Instead, we declare them as instance variables that are 'dedicated' to the appropriate method in order to achieve the simulation of the desired properties of the method instance variable. That is, the instance variables are only used by the appropriate method as a method instance variable.

The basic idea of a method instance variable though, turns out to be fairly simple: we keep all the properties of an instance variable, but the variable is now implemented and maintained at a level lower than normal in an object-oriented environment - they are now maintained inside the method.

We now have methods with variables that are not accessible by other methods of the same object of a given class, as can be seen in Figure 9. The X_miv1 and X_miv2 are accessible only from the Method_X, and Z_miv is accessible only from the Method_Z, for any given instance of class Alpha.

Thus, we can simulate having a variable that is accessible by only a single method. This variable is not accessible by other methods defined for that class. Therefore we have succeeded in eliminating the problem of having all variables accessible by all methods in a given ol ect. We have also extended the types of shared and private data in object-oriented environments.

2. Method Class Variables

Method class variables are similar in concept to method instance variables. However, method class variables are shared in value by a single method for all instances of a class, whereas method instance variables are not shared in value. Thus, we have essentially created a class variable that is limited in accessibility to a single method of the class.In our test implementation, we

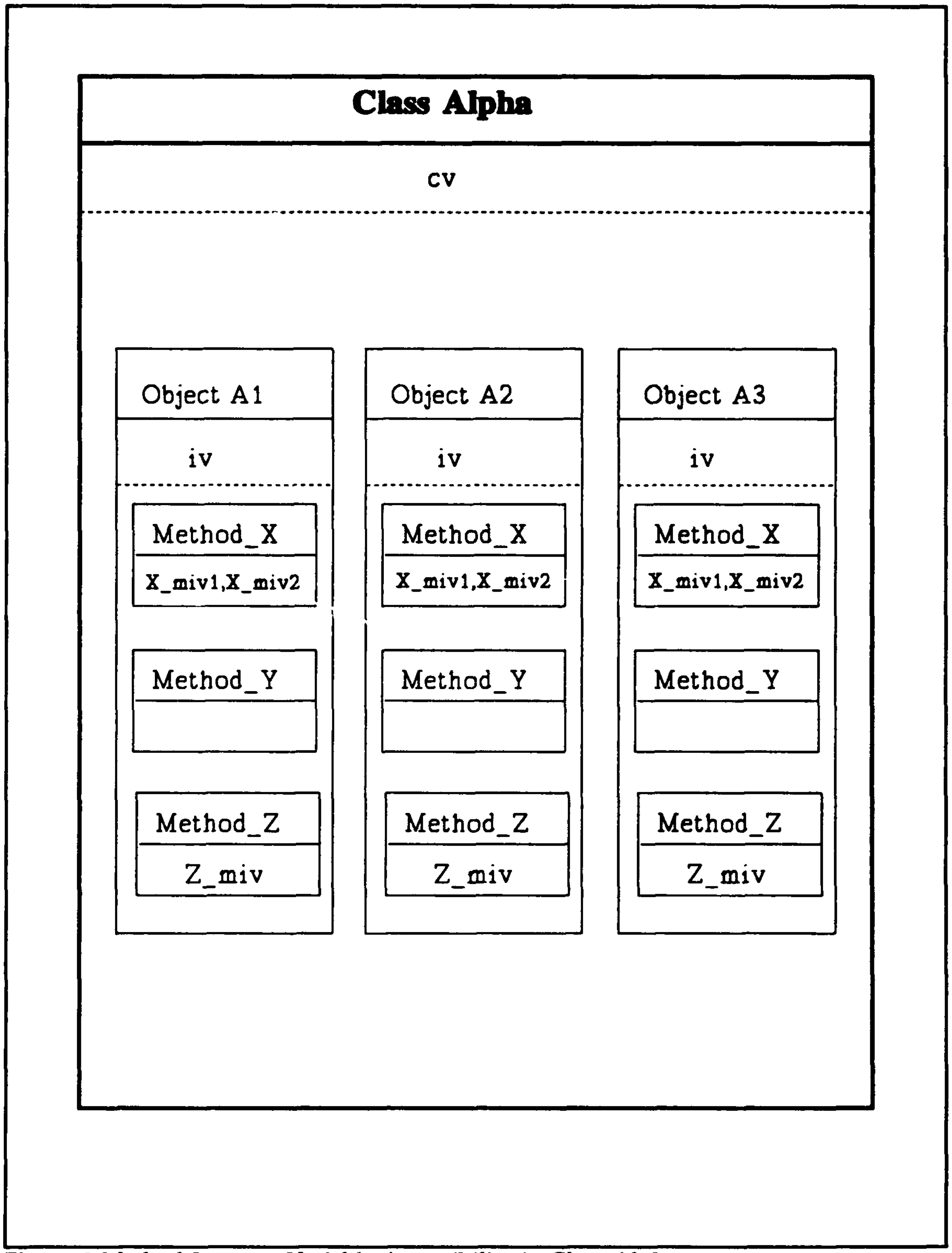

Figure 9 Method Instance Variable Accessibility in Class Alpha

create class variables which correspond to each method class variable that is defined inside a method. Once again, no actual modifications to the language itself were undertaken.

We now have methods with variables that are shared by all instances of a given class. As can be seen in Figure 10, all instances of class Alpha have a Method_X which share the method class variable X_mcv. Similarly all instances of class Alpha have a Method_Z which share the method class variable Z_mcv.

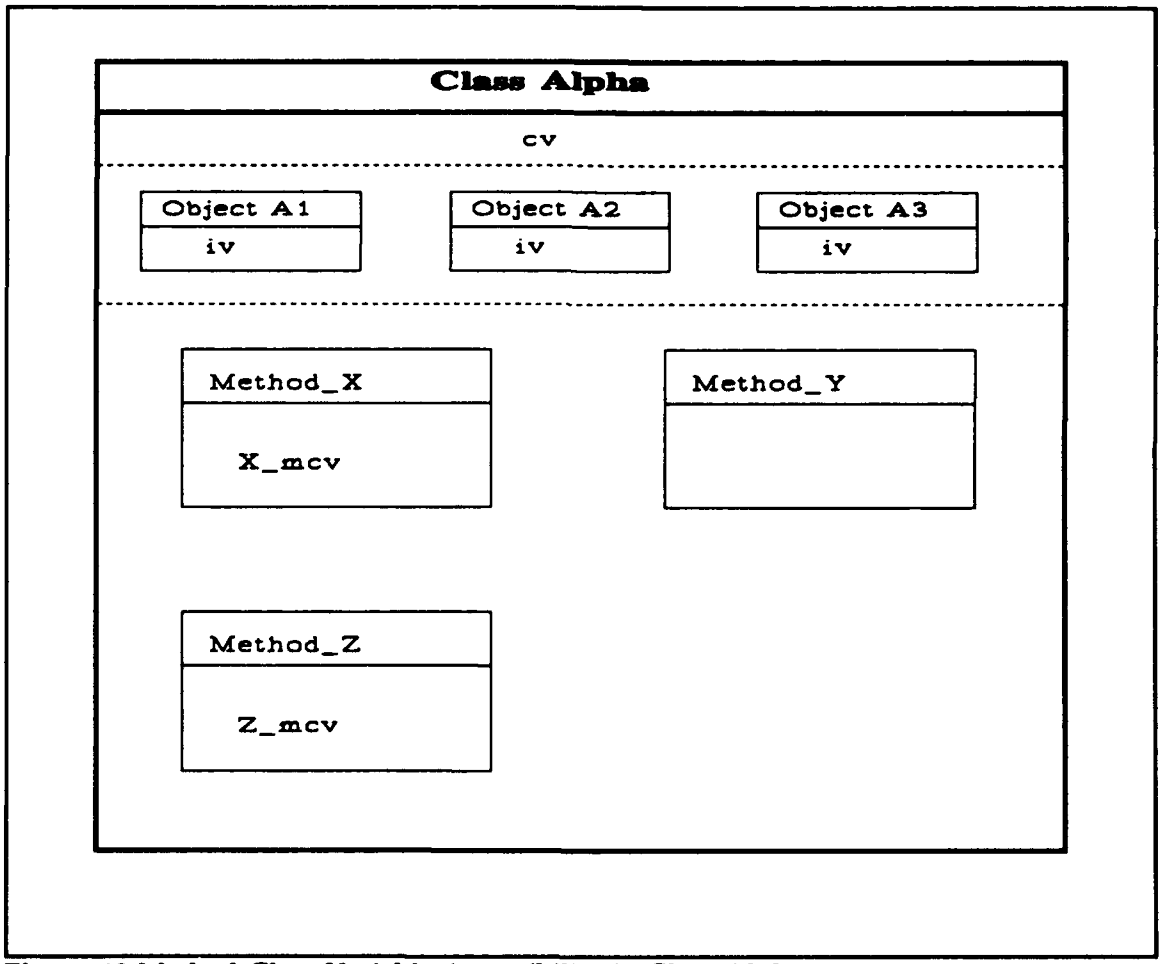

Figure 10 Method Class Variable Accessibility in Class Alpha

B. SEQUENTIAL EXECUTION

In sequential execution it is relatively easy to examine the responses of variables as they cannot be accessed by more than one process at a time. We now examine the use of method variables in a sequential environment.

1. Method Instance Variables

Following the approach used in the last chapter, we now define the class Alpha as shown in Figure 11. The purpose is to run methods in different objects using the method instance variables only for the particular method in each object. We have the methods Method_X, Method_Y, and Method_Z. Method_X has the method instance variable X_miv, and Method_Y has the method instance variable Y_miv.

The specification and implementation files of class Alpha are the *Alpha_spec_miv.ca* and *Alpha_body_miv.ca*. All files discussed in this section contained in Appendix E, Section A. In order to simulate the method instance variables, we have created instance variables that correspond to the method instance variables for each method. Thus, each time we use a method for an object, the method instance variable will still have the last value given to it when this particular method was used with the object.

The main program file is the *program_miv.ca*. Our goal here is to observe the responses of the method instance variables in different methods and objects of a class. The output of the main procedure of the program is presented

```
Class Alpha
 Superclass:          none
 Class variables:     none
 Instance variables:  none
 Methods:
  Method_X
   Method class variables:       none
   Method instance variables:    X_miv
  Method_Y
   Method class variables:       none
   Method instance variables:    Y_miv
  Method_Z
   Method class variables:       none
   Method instance variables:    none
  Create
   Method class variables:       none
   Method instance variables:    none
  Delete
   Method class variables:       none
   Method instance variables:    none
```

Figure 11 Class Alpha with Method Instance Variables

in the file *program_miv.script*. In the main procedure we create two instances of class Alpha, and apply the methods in the order Method_X, Method_Y, Method_X, Method_Y.

Each time we call a method we get the value of the instance variable corresponding to the method instance variable for that method. Note that even though we were able to declare a variable inside the method, this is actually a temporary variable that only exists while the method is executing. Also note that there is nothing that keeps other methods from accessing the instance variables maintaining the values of the method instance variables - we are only defining and testing the concept at this time, not the actual implementation[12].

As can be seen in the output, the initial value of the method instance variables was null, as no default value was given. Once the value of the method instance variable is set however, that value is still there the next time that the method is called.

As expected, the values of the method instance variables are in accordance with the observations made in Chapter III. That is, values of the method instance variables are not shared by all objects, and each time we access a method for the same object the method instance variable has the value given to it the last time this method for that object was involved. Although there is nothing in place at this time to keep other methods from accessing the instance variables maintaining the values of the method instance variables, we have simulated the ability for each method to maintain its own value for the method instance variables inside each object.

It is reasonable to expect method instance variables to behave similarly to instance variables when inheritance is considered. That is, if a method with method instance variables is inherited, then every instance of the subclass will have that method, each with its own private copy of the method instance variable.

[12]We feel that this is no different from many early object-oriented programming languages which did not provide for encapsulation of any kind. Although access to variables from outside the object was not prevented, the basic concepts were still there.

```
Class Alpha
  Superclass:          none
  Class variables:     none
  Instance variables:  none
  Methods:
    Method_X
      Method class variables:        none
      Method instance variables:     X_miv
    Method_Y
      Method class variables:        none
      Method instance variables:     Y_miv
    Method_Z
      Method class variables:        none
      Method instance variables:     none
    Create
      Method class variables:        none
      Method instance variables:     none
    Delete
      Method class variables:        none
      Method instance variables:     none

Class Beta
  Superclass:          none
  Class variables:     none
  Instance variables:  none
  Methods:
    Create
      Method class variables:        none
      Method instance variables:     none
    Delete
      Method class variables:        none
      Method instance variables:     none
```

Figure 12 The Classes Definition

Consider classes Alpha and Beta as shown in Figure 12. The class Beta inherits all variables and methods defined in the class Alpha, and the properties of those variables and methods. The files of class Beta are the *Beta_spec_miv.ca* and

Beta_body_miv.ca. All files used in this section are contained in Appendix E, Section B, except for the class Alpha files (which have not changed) that are contained in Appendix E, Section A.

In this experiment we want to verify that inherited method instance variables behave as expected. The main program is the *Program_miv_inher.ca*. and its output is contained in *Program_miv_inher.script*. Objects are created and manipulated in the following order:

- First instance of class Alpha (object A1)
- First instance of class Beta (object B1)
- Second instance of class Alpha (object A2)
- Second instance of class Beta (object B2)

We then invoke messages in the order Method_X, Method_Y, Method_X, Method_Y.

As can be seen in the output of this program, the values of the method instance variables are not shared by objects and methods. Each time we use a method in the same instance (object), it has the value given to it the last time this method was involved. Thus the, method instance variable is inherited properly by the subclass, keeping the property of not being shared between the methods of an object.

2. Method Class Variables

To illustrate the use of method class variables, we define the class Alpha as presented in Figure 13.

```
Class Alpha
 Superclass:         none
 Class variables:    none
 Instance variables: none
 Methods:
  Method_X
   Method class variables:       X_mcv
   Method instance variables:    none
  Method_Y
   Method class variables:       Y_mcv
   Method instance variables:    none
  Method_Z
  Create
   Method class variables:       none
   Method instance variables:    none
  Delete
   Method class variables:       none
   Method instance variables:    none
```

Figure 13 Class Alpha with Method Class Variables

The files with the specification and implementation of class Alpha are the *Alpha_spec_mcv.ca* and *Alpha_body_mcv.ca*. The main program, which creates instances of the class Alpha and manipulates them, is the file *program_mcv.ca*, and the output is in the file *program_mcv.script*. Our goal here is to observe the value responses of method class variables through different methods and objects of a class. The source code of all files discussed in this section are contained in Appendix F, Section A.

In the main procedure, we create two instances of class Alpha and apply methods in the order Method_X, Method_Y, Method_X, Method_Y. Each time we call a method, we get the value of the class variable used to implement of the corresponding method class variable for that method. As with method instance variables, we are only exploring and defining the concept and are not concerned with the actual implementation.

As can be seen in the output, the value of the method class variable is the one given the last time this particular method called, regardless of the object involved. Thus the value of the method class variable for each method is not shared between the methods of the object, but is shared between the same methods of the different instances in the same class.

As with method instance variables, it is reasonable to expect method class variables to behave similarly to class variables when inheritance is considered. To demonstrate this, we now define the class Beta as a subclass of Alpha, as shown in Figure 14.

The main program is the *Program_mcv_inher.ca.* and its output is contained in *Program_mcv_inher.script*. The files of class Beta are the *Beta_spec_mcv.ca* and *Beta_body_mcv.ca*. All files used in this section are contained in Appendix F, Section B except for the class Alpha files (which have not changed) that are in Appendix F, Section A. Objects are created and manipulated in the following order:

```
Class Alpha
  Superclass:          none
  Class variables:     none
  Instance variables:  none
  Methods:
    Method_X
      Method class variables:        X_mcv
      Method instance variables:     none
    Method_Y
      Method class variables:        Y_mcv
      Method instance variables:     none
    Create
      Method class variables:        none
      Method instance variables:     none
    Delete
      Method class variables:        none
      Method instance variables:     none

Class Beta
  Superclass:          Alpha
  Class variables:     none
  Instance variables:  none
  Methods:
    Create
      Method class variables:        none
      Method instance variables:     none
    Delete
      Method class variables:        none
      Method instance variables:     none
```

Figure 14 Classes Alpha and Beta Definition

- First instance of class Alpha (object A1)
- First instance of class Beta (object B1)
- Second instance of class Alpha (object A2)
- Second instance of class Beta (object B2)

Methods are then invoked in the order Method_X, Method_Y, Method_X, Method_Y. Each time we invoke a method we get the corresponding class variable for that method's method class variable.

As can be seen in the output, the value of the method class variable is the one given to it the last time this particular method was called, regardless of the object involved (or its class). Thus, the value of the method variable for each method is not shared between the methods of the instance (object), but is shared between all of the instances of a class and its subclass.

C. CONCURRENT EXECUTION

We will now show that the concept of method variables is also valid in a concurrent environment.

1. Concurrency In a Uniprocessor Environment

For these experiments we use the class Alpha as previously defined and shown in Figures 11 and 13. We want to illustrate the use of method class variables and method instance variables in a simulated concurrent environment, observing the responses of the variable.

a. Method Instance Variables

The main program, which creates instances of the class Alpha and manipulates them, is in the file *program_miv_conc.ca* and the output is in the file *program_miv_conc.script*. All files used in this section are in Appendix G, Section A. We have three tasks in the main program; t1 and t2 which create objects and send messages that invoke the methods that we want to experiment with, and the main procedure prints a simple message. To make sure that all the tasks will start running at the same time, we give the same priority to each task (pragma priority (1)). Each of the tasks t1 and t2 create an object of class Alpha and we apply the methods in the order Method_X, Method_Y, Method_X, Method_Y.

In the main task we have printed the word "main" so that we can observe when this task runs. After beginning the main program, we expect that the two tasks t1 and t2 will run in parallel with the main one. Note that before the main procedure ends, the other tasks must finished first. The final output indicated that the program executed in the following order:

- main procedure begins
- object2 created and manipulated
- object1 created and manipulated
- main procedure ends

The reason that the program chooses to run the task t2 first is arbitrary and depends only on the system, since we gave the same priority for all tasks. The results also reinforce the statement that all other tasks, must finish before the main procedure ends. As expected, the value of the method instance variable for each instance and also for each task is in accordance with the results obtained with instance variables in Chapter III.

b. *Method Class Variables*

The main program, which creates instances of the class Alpha and manipulates them, is in the file *program_mcv_conc.ca* and the output is in the file *program_mcv_conc.script*. All files used in this section are in Appendix G, Section B. We have three tasks in the main program; t1 and t2 which create objects and send messages that invoke the methods that we want to experiment with, and the main procedure prints a simple message. To make sure that all the tasks will start running at the same time, we give the same priority to each task (pragma priority (1)). Each of the tasks t1 andt2 create an object of class Alpha and apply the methods in the order Method_X, Method_Y, Method_X, Method_Y.

In the main task we have printed the word "main" so that we can observe when this task runs. After beginning the main program, we expect that the two tasks t1 and t2 will run in parallel with the main one. Note that before the main procedure ends the system has to have finished all the other tasks. The final output indicated that the program executed in the following order:

- main procedure begins
- object2 created and manipulated
- object1 created and manipulated
- main procedure ends

The reason that the program chooses to first run the task t2 first is arbitrary and depends only on the system, since we gave the same priority for all.

The results also reinforce the statement that all other tasks, must finished before the main procedure ends. As expected, the value of the method class variable for each instance and also for each task is in accordance with the results obtained with class variables in Chapter III.

2. Concurrency In a Distributed Environment

The use of method variables in a system with distributed resources, such as a Transputer, is actually similar to sequential execution. This is because Transputers do not have shared memory. Therefore, each time that the value of shared variable changes we have to update it through the communication channels to the other objects, methods, and classes. We would expect that method class variables and method instance variables would behave exactly as class variables an instance variables in this type of environment. That is, method instance variables are private for each object, and method class variables are only shared by objects residing on a single transputer.

V. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY AND CONCLUSIONS

This thesis began with a survey of the literature which served as the basis for ideas as to possible answers to our questions of how to have shared data at different levels within an object-oriented system. It then studied Classic-Ada programs to build knowledge and gain the experience of working with this object-oriented programming language in different execution environments.

The Classic-Ada programs served two main purposes. First, to study various aspects of variables in an object-oriented environment under different modes of execution. Secondly, to simulate how to implement and maintain both shared and private data at various levels in an object-oriented environment.

Our suggested solution of a new type of variable, the method variable, attempts to satisfy the following investigative questions:

1. Is it possible for a single method to have data that is not accessible by other methods defined for that class?

2. Is it possible for a single method to have data shared between various instances of a class but not accessible by other methods defined for that class?

B. RECOMMENDATIONS FOR FUTURE RESEARCH

All of the code developed for this thesis was implemented in Classic-Ada. Although this is sufficient for a general proof of concept of method variables, implementation in other object-oriented languages may be worthwhile.

More importantly though, method variables were added to an application, not to the language itself. That is, there is nothing in the language to prevent one method from accessing another method's method variables. Thus, a compiler, or at least a pre-processor, should be developed to enforce the accessibility of method variables.

Both method instance variables and method class variables are accessible by only a single method. It may also be desirable to have variables that are accessible by more than one method of a class, but not by all of them. Although this may be useful concept in an object-oriented environment, developing a clear and concise way of declaring these variables is in need of further research.

We have also not addressed the integrity of shared data in object-oriented environment. This is especially important in a concurrent or distributed system. Techniques used in conventional concurrent and distributed systems should be considered here, as should the sharing of data in database management systems.

APPENDIX A - SEQUENTIAL EXECUTION WITHOUT INHERITANCE

A. CLASS VARIABLE

1. Alpha_spec_CV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_class_variable;

   instance method Set_class_variable (temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_CV.ca

```
with text_io;
use text_io;

Class body Alpha is

 class_variable1 : Character;

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

 instance method Get_class_variable is
  begin
    put_line ("The current value of the class_variable1 at this object is:");
    put(class_variable1);
    new_line;
  end Get_class_variable;


 instance method Set_class_variable (temp_variable : in character) is
  begin
    put_line ("The new value of the class_variable1 is set and is");
    class_variable1 := temp_variable;
    put(class_variable1);
    new_line;
  end Set_class_variable;

 instance method Delete is
  begin
    put_line("Now we delete from the memory this instance of class Alpha");
    DESTROY;
  end Delete;

end Alpha;
```

3. Program_CV_one.ca

```
with Alpha;

procedure program_CV_one is

 Object1 : Object_id;
 Object2 : Object_id;
 Object3 : Object_id;

begin

  put_line("Here is the begining of the object1");
  Object1 := Alpha.Class_object;
  send (Object1, Create, new_instance => Object1);
  send (Object1, Get_class_variable);
  send (Object1, Set_class_variable, temp_variable => 'X');
  send (Object1, Get_class_variable);
  send (Object1, Delete);

  put_line("Here is the begining of the object2");
  Object2 := Alpha.Class_object;
  send (Object2, Create, new_instance => Object2);
  send (Object2, Get_class_variable);
  send (Object2, Set_class_variable, temp_variable => 'Y');
  send (Object2, Get_class_variable);
  send (Object2, Delete);

  put_line("Here is the begining of the object3");
  Object3 := Alpha.Class_object;
  send (Object3, Create, new_instance => Object3);
  send (Object3, Get_class_variable);
  send (Object3, Set_class_variable, temp_variable => 'Z' );
  send (Object3, Get_class_variable);
  send (Object3, Delete);

end program_CV_one;
```

4. Program_CV_one.script

```
Here is the beginning of the object1
in method create
The current value of the class_variable1 at this object is:

The new value of the class_variable1 is set and is :
X
The current value of the class_variable1 at this object is:
X

Now we delete from the memory this instance of class Alpha

Here is the beginning of the object2
in method create
The current value of the class_variable1 at this object is:
X
The new value of the class_variable1 is set and is :
Y
The current value of the class_variable1 at this object is:
Y

Now we delete from the memory this instance of class Alpha

Here is the beginning of the object3
in method create
The current value of the class_variable1 at this object is:
Y
The new value of the class_variable1 is set and is :
Z
The current value of the class_variable1 at this object is:
Z
Now we delete from the memory this instance of class Alpha
```

5. Program_CV_two.ca

```
with Alpha;
with text_io; use text_io;

procedure program_CV_two is

 Object1 : Object_id;
 Object2 : Object_id;
 Object3 : Object_id;

 begin

  Object1 := Alpha.Class_object;
  send (Object1, Create, new_instance => Object1);
  send (Object1, Get_class_variable);
  send (Object1, Set_class_variable, temp_variable => 'X');
  send (Object1, Get_class_variable);

  Object2 := Alpha.Class_object;
  send (Object2, Create, new_instance => Object2);
  send (Object2, Get_class_variable);
  send (Object2, Set_class_variable, temp_variable => 'Y');
  send (Object2, Get_class_variable);

  Object3 := Alpha.Class_object;
  send (Object3, Create, new_instance => Object3);
  send (Object3, Get_class_variable);
  send (Object3, Set_class_variable, temp_variable => 'Z' );
  send (Object3, Get_class_variable);


  put_line("Now we destroy the objects");
  send (Object1, Delete);
  send (Object2, Delete);
  send (Object3, Delete);

 end program_CV_two;
```

6. Program_CV_two.script

```
Here is the beginning of the object1
in method create
The current value of the class_variable1 at this object is:

The new value of the class_variable1 is set and is :
X
The current value of the class_variable1 at this object is:
X

Here is the beginning of the object2
in method create
The current value of the class_variable1 at this object is:
X
The new value of the class_variable1 is set and is :
Y
The current value of the class_variable1 at this object is:
Y

Here is the beginning of the object3
in method create
The current value of the class_variable1 at this object is:
Y
The new value of the class_variable1 is set and is :
Z
The current value of the class_variable1 at this object is:
Z


Now we destroy the objects
Now we delete from the memory this instance of class Alpha
Now we delete from the memory this instance of class Alpha
Now we delete from the memory this instance of class Alpha
```

B. INSTANCE VARIABLE

1. Alpha_spec_IV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_instance_variable;

   instance method Set_instance_variable (temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_IV.ca

```
with text_io;
use text_io;

Class body Alpha is

 instance_variable1 : instance Character;

 method Create ( new_instance : out Object_id) is
 begin
   new_instance := INSTANTIATE ;
   put_line("in method create");
 end Create;

 instance method Get_instance_variable is
 begin
   put_line ("The current value of the instance_variable1 at this object is:");
   put(instance_variable1);
   new_line;
 end Get_instance_variable;

 instance method Set_instance_variable (temp_variable: in character) is
 begin
   put_line ("The new value of the instance_variable1 is set and is :");
   instance_variable1 := temp_variable;
   put(instance_variable1);
   new_line;
 end Set_instance_variable;

 instance method Delete is
 begin
   put_line("Now we delete from the memory this instance of class Alpha");
   DESTROY;
 end Delete;

end Alpha;
```

3. Program_IV.ca

```
with Alpha;
with Beta;
with text_io; use text_io;

procedure program_IV is

  Object1 : Object_id;
  Object2 : Object_id;
  Object3 : Object_id;

begin

  put_line("Here is the beginning of the object1");
  Object1 := Alpha.Class_object;
  send (Object1, Create, new_instance => Object1);
  send (Object1, Get_instance_variable);
  send (Object1, Set_instance_variable, temp_variable => 'X');
  send (Object1, Get_instance_variable);
  new_line;
  put_line("------------------------------------");
  new_line;

  put_line("Here is the beginning of the object2");
  Object2 := Alpha.Class_object;
  send (Object2, Create, new_instance => Object2);
  send (Object2, Get_instance_variable);
  send (Object2, Set_instance_variable, temp_variable => 'Y');
  send (Object2, Get_instance_variable);
  new_line;
  put_line("------------------------------------");
  new_line;

  put_line("Here is the beginning of the object3");
  Object3 := Alpha.Class_object;
  send (Object3, Create, new_instance => Object3);
  send (Object3, Get_instance_variable);
  send (Object3, Set_instance_variable, temp_variable => 'Z' );
  send (Object3, Get_instance_variable);
  put_line("------------------------------------");
  new_line;
```

```
    put_line("Results");
    new_line;
    put_line("The value for A1 is:");
    send (Object1, Get_instance_variable);
    new_line;
    put_line("-----------------------------------");
    new_line;
    put_line("The value for A2:");
    send (Object2, Get_instance_variable);
    new_line;
    put_line("-----------------------------------");
    new_line;
    put_line("The value for A3:");
    send (Object3, Get_instance_variable);
    new_line;
    put_line("-----------------------------------");

    put_line("Now we destroy the objects");
    send (Object1, Delete);
    send (Object2, Delete);
    send (Object3, Delete);


  end program_IV;
```

4. Program_IV.script

```
Here is the beginning of the object1
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
X
The current value of the instance_variable1 at this object is:
X

------------------------------------------

Here is the beginning of the object2
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
Y
The current value of the instance_variable1 at this object is:
Y

------------------------------------------

Here is the beginning of the object3
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
Z
The current value of the instance_variable1 at this object is:
Z
------------------------------------------


Results

The value for A1 is:
The current value of the instance_variable1 at this object is:
X

------------------------------------------
```

```
The value for A2:
The current value of the instance_variable1 at this object is:
Y

-------------------------------------

The value for A3:
The current value of the instance_variable1 at this object is:
Z

-------------------------------------
Now we destroy the objects
Now we delete from the memory this instance of class Alpha
Now we delete from the memory this instance of class Alpha
Now we delete from the memory this instance of class Alpha
```

APPENDIX B - SEQUENTIAL EXECUTION WITH INHERITANCE

A. CLASS VARIABLE

1. Alpha_spec_CV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_class_variable;

   instance method Set_class_variable (temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_CV.ca

```
with text_io;
use text_io;

Class body Alpha is

 class_variable1 : Character;

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

 instance method Get_class_variable is
  begin
    put_line ("The current value of the class_variable1 at this object is:");
    put(class_variable1);
    new_line;
  end Get_class_variable;

 instance method Set_class_variable (temp_variable : in character) is
  begin
    put_line ("The new value of the class_variable1 is set ");
    put_line ("and is :");
      class_variable1 := temp_variable;
    put(class_variable1);
    new_line;
  end Set_class_variable;

 instance method Delete is
  begin
    put_line("Now we delete from the memory this instance of class Alpha");
    DESTROY;
  end Delete;

end Alpha;
```

3. Beta_spec_CV.ca

```
class Beta is
  superclass Alpha;
   method Create (New_Instance : out Object_id );
   instance method Delete;
end Beta;
```

4. Beta_body_CV.ca

```
with text_io;
use text_io;

Class body Beta is

 method Create ( new_instance : out Object_id) is
   begin
     new_instance := INSTANTIATE ;
     put_line("in method create");
   end Create;

 instance method Delete is
   begin
     put_line("Now we delete from the memory this instance of class Beta");
     send ( super, Delete);
     DESTROY;
   end Delete;

end Beta;
```

5. Program_CV_inher.ca

```
with Alpha;
with Beta;
with text_io; use text_io;

procedure program_CV_inher is

  ObjectA1 : Object_id;
  ObjectA2 : Object_id;
  ObjectB1 : Object_id;
  ObjectB2 : Object_id;

begin

  new_line;
  put_line("Here is the beginning of object A1");
  new_line;
  ObjectA1 := Alpha.Class_object;
  send (ObjectA1, Create, new_instance => ObjectA1);
  send (ObjectA1, Get_class_variable);
  send (ObjectA1, Set_class_variable , temp_variable => 'X');
  send (ObjectA1, Get_class_variable);
  new_line;
  put_line("------------------------------------");
  new_line;

  put_line("Here is the beginning of object B1");
  new_line;
  ObjectB1 := Beta.Class_object;
  send (ObjectB1, Create, new_instance => ObjectB1);
  send (ObjectB1, Get_class_variable);
  send (ObjectB1, Set_class_variable , temp_variable => 'M');
  send (ObjectB1, Get_class_variable);
  new_line;
  put_line("------------------------------------");
  new_line;
```

```
new_line;
put_line("Here is the beginning of object A2");
mew_line;
ObjectA2 := Alpha.Class_object;
send (ObjectA2, Create, new_instance => ObjectA2);
send (ObjectA2, Get_class_variable);
send (ObjectA2, Set_class_variable , temp_variable => 'Y');
send (ObjectA2, Get_class_variable);
new_line;
put_line("----------------------------------------");
new_line;

new_line;
put_line("Here is the beginning of object B2");
mew_line;
objectb2 := Beta.Class_object;
send (ObjectB2, Create, new_instance => ObjectB2);
send (ObjectB2, Get_class_variable);
send (ObjectB2, Set_class_variable , temp_variable => 'N');
send (ObjectB2, Get_class_variable);
new_line;
put_line("----------------------------------------");
new_line;


put_line("-------------Total Results---------------");
new_line;
put_line("The value of  A1:");
send (ObjectA1, Get_class_variable);
put_line("The value of  A2:");
send (ObjectA2, Get_class_variable);
put_line("The value of  B1:");
send (ObjectB1, Get_class_variable);
put_line("The value of  B2:");
send (ObjectB2, Get_class_variable);
new_line;

put_line("for A1");
send (ObjectA1, Delete);

put_line("for A2");
```

```
  send (ObjectA2, Delete);


  put_line("for B1");
  send (ObjectB1, Delete);

  put_line("for B2");
  send (ObjectB2, Delete);

end program_CV_inher;
```

6. Program_CV_inher.script

```
Here is the beginning of object A1
in method create
The current value of the class_variable1 at this object is:

The new value of the class_variable1 is set and is :
X
The current value of the class_variable1 at this object is:
X
------------------------------------

Here is the beginning of object B1
in method create
The current value of the class_variable1 at this object is:
X
The new value of the class_variable1 is set and is :
M
The current value of the class_variable1 at this object is:
M
---------------------------------

Here is the beginning of the object A2
in method create
The current value of the class_variable1 at this object is:
M
The new value of the class_variable1 is set and is :
Y
The current value of the class_variable1 at this object is:
Y
---------------------------------

Here is the beginning of object B2
in method create
The current value of the class_variable1 at this object is:
Y
The new value of the class_variable1 is set and is :
N
The current value of the class_variable1 at this object is:
N
-----------------------------------
```

```
------------Total Results------------

The value of  A1:
The current value of the class_variable1 at this object is:
N
The value of  A2:
The current value of the class_variable1 at this object is:
N
The value of  B1:
The current value of the class_variable1 at this object is:
N
The value of  B2:
The current value of the class_variable1 at this object is:
N

for A1
Now we delete from the memory this instance of class Alpha
for A2
Now we delete from the memory this instance of class Alpha
for B1
Now we delete from the memory this instance of class Beta
Now we delete from the memory this instance of class Alpha
for B2
Now we delete from the memory this instance of class Beta
Now we delete from the memory this instance of class Alpha
```

B. INSTANCE VARIABLE

1. Alpha_spec_IV.ca

```
class Alpha is

    method Create (New_Instance : out Object_id );

    instance method Get_instance_variable;

    instance method Set_instance_variable (temp_variable : in character);

    instance method Delete;

end Alpha;
```

2. Alpha_body_IV.ca

```
with text_io;
use text_io;

Class body Alpha is

 instance_variable1 : instance Character;

 method Create ( new_instance : out Object_id) is
  begin
   new_instance := INSTANTIATE ;
   put_line("in method create");
  end Create;

 instance method Get_instance_variable is
  begin
   put_line ("The current value of the instance_variable1 at this object is:");
   put(instance_variable1);
   new_line;
  end Get_instance_variable;

 instance method Set_instance_variable (temp_variable: in character) is
  begin
   put_line ("The new value of the instance_variable1 is set and is :");
   instance_variable1 := temp_variable;
   put(instance_variable1);
   new_line;
  end Set_instance_variable;

 instance method Delete is
  begin
   put_line("Now we delete from the memory this instance of class Alpha");
   DESTROY;
  end Delete;

end Alpha;
```

3. Beta_spec_IV.ca

```
class Beta is
   superclass Alpha;
    method Create (New_Instance : out Object_id );
   instance method Delete;
end Beta;
```

4. Beta_body_IV.ca

```
with text_io;
use text_io;

Class body Beta is

 method Create ( new_instance : out Object_id) is
   begin
     new_instance := INSTANTIATE ;
     put_line("in method create");
   end Create;

 instance method Delete is
   begin
     put_line("Now we delete from the memory this instance of class Beta");
     send ( super, Delete);
     DESTROY;
   end Delete;

end Beta;
```

5. Program_IV_inher.ca

```
with Alpha;
with Beta;
with text_io;
use text_io;

procedure program_IV_inher is

  ObjectA1 : Object_id;
  ObjectA2 : Object_id;
  ObjectB1 : Object_id;
  ObjectB2 : Object_id;

begin

  new_line;
  put_line("Here is the beginning of object A1");
  new_line;
  ObjectA1 := Alpha.Class_object;
  send (ObjectA1, Create, new_instance => ObjectA1);
  send (ObjectA1, Get_instance_variable);
  send (ObjectA1, Set_instance_variable, temp_variable => 'X' );
  send (ObjectA1, Get_instance_variable);
  new_line;
  put_line("----------------------------------------");
  new_line;

  put_line("Here is the beginning of object B1");
  ObjectB1 := Beta.Class_object;
  send (ObjectB1, Create, new_instance => ObjectB1);
  send (ObjectB1, Get_instance_variable);
  send (ObjectB1, Set_instance_variable , temp_variable => 'M' );
  send (ObjectB1, Get_instance_variable);
  new_line;
  put_line("----------------------------------------");
  new_line;

  put_line("Here is the beginning of the object A2");
  ObjectA2 := Alpha.Class_object;
  send (ObjectA2, Create, new_instance => ObjectA2);
  send (ObjectA2, Get_instance_variable);
  send (ObjectA2, Set_instance_variable , temp_variable => 'Y' );
  send (ObjectA2, Get_instance_variable);
  new_line;
  put_line("------------------------------");
  new_line;

  put_line("Here is the beginning of object B2");
  objectb2 := Beta.Class_object;
  send (ObjectB2, Create, new_instance => ObjectB2);
  send (ObjectB2, Get_instance_variable);
  send (ObjectB2, Set_instance_variable , temp_variable => 'N');
  send (ObjectB2, Get_instance_variable );
  new_line;
  put_line("--------------------------------");
  new_line;

  put_line("------------Total Results-------------");
  new_line;
  put_line("The value of  A1:");
  send (ObjectA1, Get_instance_variable);
  put_line("The value of  A2:");
  send (ObjectA2, Get_instance_variable);
  put_line("The value of  B1:");
  send (ObjectB1, Get_instance_variable);
  put_line("The value of  B2:");
  send (ObjectB2, Get_instance_variable);
  new_line;

  put_line("for A1");
  send (ObjectA1, Delete);

  put_line("for A2");
  send (ObjectA2, Delete);

  put_line("for B1");
  send (ObjectB1, Delete);

  put_line("for B2");
  send (ObjectB2, Delete);

end program_IV_inher;
```

6. Program_IV_inher.script

```
Here is the beginning of object A1
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
X
The current value of the instance_variable1 at this object is:
X
-----------------------------------------

Here is the beginning of object B1
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
M
The current value of the instance_variable1 at this object is:
M
-------------------------------------

Here is the beginning of the object A2
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
Y
The current value of the instance_variable1 at this object is:
Y
-------------------------------------

Here is the beginning of object B2
in method create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
N
The current value of the instance_variable1 at this object is:
N
---------------------------------------
```

```
------------Total Results-------------

The value of  A1:
The current value of the instance_variable1 at this object is:
X
The value of  A2:
The current value of the instance_variable1 at this object is:
Y
The value of  B1:
The current value of the instance_variable1 at this object is:
M
The value of  B2:
The current value of the instance_variable1 at this object is:
N

for A1
Now we delete from the memory this instance of class Alpha
for A2
Now we delete from the memory this instance of class Alpha
for B1
Now we delete from the memory this instance of class Beta
Now we delete from the memory this instance of class Alpha
for B2
Now we delete from the memory this instance of class Beta
Now we delete from the memory this instance of class Alpha
```

APPENDIX C - CONCURRENCY ON A UNIPROCESSOR

A. CLASS VARIABLE

1. Alpha_spec_CV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_class_variable;

   instance method Set_class_variable (temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_CV.ca

```
with text_io;
use text_io;

Class body Alpha is

 class_variable1 : Character;

 method Create ( new_instance : out Object_id) is
   begin
     new_instance := INSTANTIATE ;
     put_line("in method create");
   end Create;

 instance method Get_class_variable is
   begin
     put_line ("The current value of the class_variable1 at this object is:");
     put(class_variable1);
     new_line;
   end Get_class_variable;


 instance method Set_class_variable (temp_variable : in character) is
   begin
     put_line ("The new value of the class_variable1 is set ");
     put_line ("and is :");
       class_variable1 := temp_variable1;
     put(class_variable1);
     new_line;
   end Set_class_variable;

 instance method Delete is
   begin
     put_line("Now we delete from the memory this instance of class Alpha");
     DESTROY;
   end Delete;

end Alpha;
```

3. Program_CV_conc.ca

```
with Alpha;
with text_io;
use text_io;

procedure program_CV_conc is

 pragma priority (1);

 task t1 is
  pragma priority(1);
 end;

 task body t1 is
  Object1 :Object_id;
  begin
    put_line("Here is the object A1");
    Object1 := Alpha.Class_object;
    send (Object1, Create, new_instance => Object1);
    put_line("in the first object after create");
    send (Object1, Get_class_variable);
    send (Object1, Set_class_variable, temp_variable =>'X');
    send (Object1, Get_class_variable);
    new_line;
    put_line("----------------------------------------");
    new_line;
  end t1;

 task t2 is
  pragma priority(1);
 end;

 task body t2 is
  Object2 : Object_id;
  begin
    put_line("Here is the beginning of object A2");
    Object2 := Alpha.Class_object;
    send (Object2, Create, new_instance => Object2 );
    put_line("in the A2 object after create");
    send (Object2, Get_class_variable);
    send (Object2, Set_class_variable, temp_variable=>'Y');
```

```
   send (Object2, Get_class_variable);
   new_line;
   put_line("------------------------------");
   new_line;
 end t2;

task t3 is
 pragma priority(1);
end;

task body t3 is
 Object3 : Object_id;
 begin
   put_line("Here is the beginning of the A3 object");
   Object3 := Alpha.Class_object;
   send ( Object3, Create, new_instance => Object3 );
   put_line("in the third object after create");
   send (Object3, Get_class_variable);
   send (Object3, Set_class_variable , temp_variable=> 'Z');
   send (Object3, Get_class_variable);
   new_line;
   put_line("---------------------------------");
   new_line;
 end t3;

begin
  put_line("main");
   put_line("we are going do delete Object1");
   send (Object1, Delete);
   put_line("we deleted object1");
   send (Object2, Delete);
   put_line("we deleted object2");
   send (Object3, Delete);
   put_line("we deleted object3");
end program_CV_conc;
```

4. Program_CV_conc.script

```
Here is the beginning of the A3 object
in method create
in the third object after create
The current value of the class_variable1 at this object is:

The new value of the class_variable1 is set and is :
Z
The current value of the class_variable1 at this object is:
Z
-----------------------------------

Here is the beginning of object A2
in method create
in the A2 object after create
The current value of the class_variable1 at this object is:
Z
The new value of the class_variable1 is set and is :
Y
The current value of the class_variable1 at this object is:
Y
-----------------------------------

Here is the object A1
in method create
in the first object after create
The current value of the class_variable1 at this object is:
Y
The new value of the class_variable1 is set and is :
X
The current value of the class_variable1 at this object is:
X
-----------------------------------

main
we are going do delete Object1
Now we delete from the memory this instance of class Alpha
we deleted object1
Now we delete from the memory this instance of class Alpha
we deleted object2
Now we delete from the memory this instance of class Alpha
we deleted object3
```

B. INSTANCE VARIABLE

1. Alpha_spec_IV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_instance_variable;

   instance method Set_instance_variable (temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_IV.ca

```
with text_io;
use text_io;

Class body Alpha is

 instance_variable1 : instance Character;

 method Create ( new_instance : out Object_id) is
  begin
   new_instance := INSTANTIATE ;
   put_line("in method create");
  end Create;

 instance method Get_instance_variable is
  begin
   put_line ("The current value of the instance_variable1 at this object is:");
   put(instance_variable1);
   new_line;
  end Get_instance_variable;

  instance method Set_instance_variable (temp_variable: in character) is
   begin
    put_line ("The new value of the instance_variable1 is set and is :");
    instance_variable1 := temp_variable;
    put(instance_variable1);
    new_line;
   end Set_instance_variable;

   instance method Delete is
    begin
     put_line("Now we delete from the memory this instance of class Alpha");
     DESTROY;
    end Delete;

end Alpha;
```

3. Program_IV_conc.ca

```
with Alpha;
with text_io; use text_io;

procedure program_IV_conc is

 Object1 :Object_id;
 Object2 : Object_id;
 Object3 : Object_id;

 pragma priority (1);

 task t1 is
  pragma priority(1);
 end;

 task body t1 is
  begin
    put_line("Here is the object A1");
    Object1 := Alpha.Class_object;
    send (Object1, Create, new_instance => Object1);
    put_line("in the first object after create");
    send (Object1, Get_instance_variable);
    send (Object1, Set_instance_variable, temp_variable =>'X');
    send (Object1, Get_instance_variable);
    new_line;
    put_line("---------------------------------");
    new_line;
  end t1;

 task t2 is
  pragma priority(1);
 end;

 task body t2 is
  begin
    put_line("Here is the beginning of object A2");
    Object2 := Alpha.Class_object;
    send (Object2, Create, new_instance => Object2 );
    put_line("in the A2 object after create");
    send (Object2, Get_instance_variable);
  send (Object2, Set_instance_variable, temp_variable=>'Y');
  send (Object2, Get_instance_variable);
  new_line;
  put_line("------------------------------");
  new_line;
 end t2;

task t3 is
 pragma priority(1);
end;

task body t3 is
 begin
  put_line("Here is the beginning of the A3 object");
  Object3 := Alpha.Class_object;
  send ( Object3, Create, new_instance => Object3 );
  put_line("in the third object after create");
  send (Object3, Get_instance_variable);
  send (Object3, Set_instance_variable , temp_variable=> 'Z');
  send (Object3, Get_instance_variable);
  new_line;
  put_line("------------------------------");
  new_line;
 end t3;

 begin
  put_line("main");
  put_line("we are going do delete Object1");
  send (Object1, Delete);
  put_line("we deleted object1");
  send (Object2, Delete);
  put_line("we deleted object2");
  send (Object3, Delete);
  put_line("we deleted object3");
 end program_IV_conc;
```

4. Program_IV_conc.script

```
Here is the beginning of the A3 object
in method create
in the third object after create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
Z
The current value of the instance_variable1 at this object is:
Z
-----------------------------------

Here is the beginning of object A2
in method create
in the A2 object after create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
Y
The current value of the instance_variable1 at this object is:
Y
-----------------------------------

Here is the object A1
in method create
in the first object after create
The current value of the instance_variable1 at this object is:

The new value of the instance_variable1 is set and is :
X
The current value of the instance_variable1 at this object is:
X
-----------------------------------
main
we are going do delete Object1
Now we delete from the memory this instance of class Alpha
we deleted object1
Now we delete from the memory this instance of class Alpha
we deleted object2
Now we delete from the memory this instance of class Alpha
we deleted object3
```

APPENDIX D - CONCURRENCY ON A MULTIPROCESSOR

A. TWO TRANSPUTERS WITH CLASS VARIABLE

1. Alpha_spec_CV.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Get_class_variable;

   instance method Set_class_variable (class_variable1 : in out integer);

   instance method Delete;

end Alpha;
```

2. Alpha_body_CV.ca

```
with text_io;
use text_io;

Class body Alpha is
 package integer_inout is new integer_io(integer);
 use integer_inout;
 class_variable1 : Integer;
 Value: integer;

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

  instance method Get_class_variable is
   begin
     put_line ("The current value of the class_variable1 at this object is:");
     put(class_variable1);
     new_line;
   end Get_class_variable;

   instance method Set_class_variable (class_variable1: in out integer) is
    begin
      put_line ("The new value of the class_variable1 is set ");
      put_line ("and is :");
      put(class_variable1);
      new_line;
    end Set_class_variable;

   instance method Delete is
    begin
      put_line("Now we delete from the memory this instance of class Alpha");
      DESTROY;
    end Delete;

end Alpha;
```

3. Program_CV_trans.ca

```
with Alpha;
use Alpha;

procedure proj is

 class_variable:integer;
 one :  integer:=1;
 two :  integer:=2;
 Object1 : Object_id;
 Object2 : Object_id;

 begin

   Object1 := Alpha.Class_object;
   send (Object1, Create, new_instance => Object1);
   send (Object1, Get_class_variable);
   send (Object1, Set_class_variable, class_variable1 => one);
   send (Object1, Get_class_variable);
   send (Object1, Delete);

   Object2 := Alpha.Class_object;
   send (Object2, Create, new_instance => Object2);
   send (Object2, Get_class_variable);
   send (Object2, Set_class_variable, class_variable1 => two);
   send (Object2, Get_class_variable);
   send (Object2, Delete);
 end proj;
```

4. Alpha_one.ada

```
WITH Classic_Executive;   USE Classic_Executive;
WITH Unchecked_Deallocation;
WITH Unchecked_Conversion;
WITH System;
with Alpha;
with COMMON;
use COMMON;
with CHANNELS;

procedure Alpha_one is

C : CHANNELS.CHANNEL_REF := CHANNELS.OUT_PARAMETERS (2);

   RESULT : int_16;
   Object2 : Object_id;

begin
  Object2 := Alpha.Class_object;

    DECLARE
     TYPE Parameter_Type IS RECORD
        New_Instance : Object_id;
      END RECORD;

      Parameter_Data : Parameter_Type;

    BEGIN
      send ( Object2, 1, Assign (Parameter_Data'Address) );
      Object2 := Parameter_Data.New_Instance;
    END;

    RESULT :=2;

   INTEGER_IO.write(C, RESULT);
    send ( Object2, 4 );


 end Alpha_one;
```

5. Alpha_two.ada

```
WITH Classic_Executive;   USE Classic_Executive;
WITH Unchecked_Deallocation;
WITH Unchecked_Conversion;
WITH System;
with Alpha;
with text_io;
with COMMON;
use COMMON;
with CHANNELS;

procedure Alpha_two is


C : CHANNELS.CHANNEL_REF := CHANNELS.IN_PARAMETERS (2);

result: int_16;
  Object1 : Object_id;

begin
  Object1 := Alpha.Class_object;

    DECLARE

      TYPE Parameter_Type IS RECORD
        New_Instance : Object_id;
      END RECORD;

      Parameter_Data : Parameter_Type;

    BEGIN
      send ( Object1, 1, Assign (Parameter_Data'Address) );
      Object1 := Parameter_Data.New_Instance;
    END;

    send ( Object1, 2 );

    send ( Object1, 3 );
    text_io.put("I am going to read");
     integer_io.read(C, result);

   text_io.put_line(int_16'IMAGE(result));

    send ( Object1, 2 );

    send ( Object1, 4 );


end Alpha_two;
```

6. Program_CV_trans.script

```
in method create
The current value of the class_variable1 at this object is:
        0
The new value of the class_variable1 is set
and is :
        1
I am going to read  2
The current value of the class_varaible1 at this object is:
        1
Now we delete from the memory this instance of class Alpha
```

APPENDIX E - METHOD INSTANCE VARIABLES

A. CLASSES WITHOUT INHERITANCE RELATION

1. Alpha_spec_miv.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Method_X(temp_variable : in character);

   instance method Method_Y(temp_variable : in character);

   instance method Method_Z;

   instance method Delete;

end Alpha;
```

2. Alpha_body_miv.ca

```
with text_io;
use text_io;

Class body Alpha is

instance_variable1 : instance Character;
X_im :instance Character;
Y_im :instance Character;

method Create ( new_instance : out Object_id) is

  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

instance method Method_X(temp_variable: in character) is
  X_miv : Character;
  begin
    X_miv := X_im;
    put_line ("The value of X_miv in this method is:");
    new_line;
    put(X_miv);
    new_line;
    X_miv := temp_variable;
    put_line ("The new value of X_miv in this method is set and is:");
    put(X_miv);
    new_line;
    X_im := X_miv;
  end Method_X;

instance method Method_Y(temp_variable: in character) is
  Y_miv : Character;
  begin
    Y_miv := Y_im;
    put_line ("The value of Y_miv in this method is:");
    new_line;
    put(Y_miv);
    new_line;
    Y_miv := temp_variable;
    put_line ("The new value of Y_miv in this method is set and is:");
    put(Y_miv);
    new_line;
    Y_im :=Y_miv;
  end Method_Y;

 instance method Method_Z  is
  begin
    new_line;
  end Method_Z;

 instance method Delete is
  begin
    DESTROY;
  end Delete;

end Alpha;
```

3. **Program_miv.ca**

```
with Alpha;
with text_io; use text_io;

procedure  program_miv  is

 Object1 : Object_id;
 Object2 : Object_id;

 begin
   put_line("Here is the beginning of the object1");
   Object1 := Alpha.Class_object;
   send (Object1, Create, new_instance => Object1);
   put_line("Here is the Method_X of the object1");
   send (Object1, Method_X, temp_variable => 'X');
   put_line("----------------------------------");
   put_line("Here is the Method_Y of the object1");
   send (Object1,Method_Y , temp_variable => 'Y');
   put_line("----------------------------------");
   put_line("Here is the Method_X of the object1");
   send (Object1, Method_X, temp_variable => 'Z');
   put_line("----------------------------------");
   put_line("Here is the Method_Y of the object1");
   send (Object1,Method_Y , temp_variable => 'W');
   put_line("----------------------------------");
   new_line;
   put_line("End of A1----------------------------------");
   new_line;

   put_line("Here is the beginning of the object2");
   Object2 := Alpha.Class_object;
   send (Object2, Create, new_instance => Object2);
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'K');
   put_line("----------------------------------");
   put_line("Here is the Method_Y of the object2");
   send (Object2,Method_Y , temp_variable => 'L');
   put_line("----------------------------------");
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'M');
   put_line("----------------------------------");
```

```
  put_line("Here is the Method_Y of the object2");
  send (Object2,Method_Y , temp_variable => 'N');
  put_line("-----------------------------------");
  new_line;
  put_line("End of A2-----------------------------------");
  new_line;

  put_line("Now we destroy the objects");
  send (Object1, Delete);
  send (Object2, Delete);

end program_miv;
```

4. Program_miv.script

```
Here is the beginning of the object1
in method create
Here is the Method_X of the object1
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
X

----------------------------------------
Here is the Method_Y of the object1
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
Y

----------------------------------------
Here is the Method_X of the object1
The value of X_miv in this method is:
X
The new value of X_miv in this method is set and is:
Z

----------------------------------------
Here is the Method_Y of the object1
The value of Y_miv in this method is:
Y
The new value of Y_miv in this method is set and is:
W

----------------------------------------
End of A1----------------------------------------


Here is the beginning of the object2
in method create
Here is the Method_X of the object2
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
K
```

```
-----------------------------------
Here is the Method_Y of the object2
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
L

-----------------------------------
Here is the Method_X of the object2
The value of X_miv in this method is:
K
The new value of X_miv in this method is set and is:
M

-----------------------------------
Here is the Method_Y of the object2
The value of Y_miv in this method is:
L
The new value of Y_miv in this method is set and is:
N

-----------------------------------
End of A2-----------------------------------
```

B. CLASSES WITH INHERITANCE RELATION

1. Beta_spec_miv.ca

```
class Beta is
   superclass Alpha;

   method Create (New_Instance : out Object_id );

   instance method Delete;

end Beta;
```

2. Beta_body_miv.ca

```
with text_io;
use text_io;

Class body Beta is

 method Create ( new_instance : out Object_id) is

  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

  instance method Delete is
  begin
   send ( super, Delete);
   DESTROY;
  end Delete;

end Beta;
```

3. Program_miv_inher.ca

```
with Alpha;
with Beta;
with text_io; use text_io;

procedure program_miv_inher is

ObjectA1 : Object_id;
ObjectA2 : Object_id;
ObjectB1 : Object_id;
ObjectB2 : Object_id;

begin

  put_line("Here is the beginning of the objectA1");
  ObjectA1 := Alpha.Class_object;
  send (ObjectA1, Create, new_instance => ObjectA1);
  put_line("Here is the Method_X of the objectA1");
  send (ObjectA1, Method_X, temp_variable => 'X');
  put_line("----------------------------------");
  put_line("Here is the Method_Y of the objectA1");
  send (ObjectA1,Method_Y , temp_variable => 'Y');
  put_line("----------------------------------");
  put_line("Here is the Method_X of the objectA1");
  send (ObjectA1, Method_X, temp_variable => 'Z');
  put_line("----------------------------------");
  put_line("Here is the Method_Y of the objectA1");
  send (ObjectA1,Method_Y , temp_variable => 'W');
  put_line("----------------------------------");
  new_line;
  put_line("End of A1----------------------------------");
  new_line;

  put_line("Here is the beginning of the objectB1");
  ObjectB1 := Alpha.Class_object;
  send (ObjectB1, Create, new_instance => ObjectB1);
  put_line("Here is the Method_X of the objectB1");
  send (ObjectB1, Method_X, temp_variable => 'X');
  put_line("----------------------------------");
  put_line("Here is the Method_Y of the objectB1");
  send (ObjectB1,Method_Y , temp_variable => 'Y');
  put_line("----------------------------------");
put_line("Here is the Method_X of the objectB1");
send (ObjectB1, Method_X, temp_variable => 'Z');
put_line("-----------------------------------");
put_line("Here is the Method_Y of the objectB1");
send (ObjectB1,Method_Y , temp_variable => 'W');
put_line("-----------------------------------");
new_line;
put_line("End of B1-----------------------------------");
new_line;


put_line("Here is the beginning of the objectA2");
ObjectA2 := Alpha.Class_object;
send (ObjectA2, Create, new_instance => ObjectA2);
put_line("Here is the Method_X of the objectA2");
send (ObjectA2, Method_X, temp_variable => 'K');
put_line("-----------------------------------");
put_line("Here is the Method_Y of the objectA2");
send (ObjectA2,Method_Y , temp_variable => 'L');
put_line("-----------------------------------");
put_line("Here is the Method_X of the objectA2");
send (ObjectA2, Method_X, temp_variable => 'M');
put_line("-----------------------------------");
put_line("Here is the Method_Y of the objectA2");
send (ObjectA2,Method_Y , temp_variable => 'N');
put_line("-----------------------------------");
new_line;
put_line("End of A2-----------------------------------");
new_line;

put_line("Here is the beginning of the objectB2");
ObjectB2 := Alpha.Class_object;
send (ObjectB2, Create, new_instance => ObjectB2);
put_line("Here is the Method_X of the objectB2");
send (ObjectB2, Method_X, temp_variable => 'K');
put_line("-----------------------------------");
put_line("Here is the Method_Y of the objectB2");
send (ObjectB2,Method_Y , temp_variable => 'L');
put_line("-----------------------------------");
put_line("Here is the Method_X of the objectB2");
send (ObjectB2, Method_X, temp_variable => 'M');
put_line("-----------------------------------");
put_line("Here is the Method_Y of the objectB2");
```

```
  send (ObjectB2,Method_Y , temp_variable => 'N');
  put_line("------------------------------------");
  new_line;
  put_line("End of B2------------------------------------");
  new_line;

  put_line("Now we destroy the objects");
  send (ObjectA1, Delete);
  send (ObjectA2, Delete);
  send (ObjectB1, Delete);
  send (ObjectB2, Delete);

end program_miv_inher;
```

4. Program_miv_inher.script

```
Here is the beginning of the objectA1
in method create
Here is the Method_X of the objectA1
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
X
-----------------------------------
Here is the Method_Y of the objectA1
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
Y
-----------------------------------
Here is the Method_X of the objectA1
The value of X_miv in this method is:
X
The new value of X_miv in this method is set and is:
Z
-----------------------------------
Here is the Method_Y of the objectA1
The value of Y_miv in this method is:
Y
The new value of Y_miv in this method is set and is:
W
-----------------------------------

End of A1-----------------------------------

Here is the beginning of the objectB1
in method create
Here is the Method_X of the objectB1
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
X
-----------------------------------
Here is the Method_Y of the objectB1
The value of Y_miv in this method is:
```

```
The new value of Y_miv in this method is set and is:
Y
----------------------------------------
Here is the Method_X of the objectB1
The value of X_miv in this method is:
X
The new value of X_miv in this method is set and is:
Z
----------------------------------------
Here is the Method_Y of the objectB1
The value of Y_miv in this method is:
Y
The new value of Y_miv in this method is set and is:
W
----------------------------------------

End of B1----------------------------------------

Here is the beginning of the objectA2
in method create
Here is the Method_X of the objectA2
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
K
----------------------------------------
Here is the Method_Y of the objectA2
The value of Y_miv in this method is:

The new value of X_miv in this method is set and is:
L
----------------------------------------
Here is the Method_X of the objectA2
The value of X_miv in this method is:
K
The new value of X_miv in this method is set and is:
M
----------------------------------------
Here is the Method_Y of the objectA2
The value of Y_miv in this method is:
L
```

```
The new value of X_miv in this method is set and is:
N
-----------------------------------

End of A2-----------------------------------

Here is the beginning of the objectB2
in method create
Here is the Method_X of the objectB2
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
X
-----------------------------------
Here is the Method_Y of the objectB2
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
Y
-----------------------------------
Here is the Method_X of the objectB2
The value of X_miv in this method is:
X
The new value of X_miv in this method is set and is:
Z
-----------------------------------
Here is the Method_Y of the objectB2
The value of Y_miv in this method is:
Y
The new value of Y_miv in this method is set and is:
W
-----------------------------------

End of B2-----------------------------------

Now we destroy the objects
```

APPENDIX F - METHOD CLASS VARIABLES

A. CLASSES WITHOUT INHERITANCE

1. Alpha_spec_mcv.ca

```
class Alpha is

   method Create (New_Instance : out Object_id );

   instance method Method_X(temp_variable : in character);

   instance method Method_Y(temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_mcv.ca

```
with text_io;
use text_io;

Class body Alpha is

 X_mcv : Character;
 Y_mcv : Character;

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

 instance method Method_X(temp_variable: in character) is
  begin
    put_line ("The value of X_mcv in this method is:");
    new_line;
    put(X_mcv);
    new_line;
    X_mcv := temp_variable;
    put_line ("The new value of mv in this method is set and is:");
    put(X_mcv);
    new_line;
  end Method_X;

 instance method Method_Y(temp_variable: in character) is
  begin
    put_line ("The value of Y_mcv in this method is:");
    new_line;
    put(Y_mcv);
    new_line;
    Y_mcv := temp_variable;
    put_line ("The new value of mv in this method is set and is:");
    put(Y_mcv);
    new_line;
  end Method_Y;
  instance method Delete is
   begin
     DESTROY;
   end Delete;

end Alpha;
```

3. Program_mcv.ca

```
with Alpha;
with text_io; use text_io;

procedure  program_mcv  is

 Object1 : Object_id;
 Object2 : Object_id;

begin

  put_line("Here is the beginning of the object1");
  Object1 := Alpha.Class_object;
  send (Object1, Create, new_instance => Object1);
  put_line("Here is the Method_X of the object1");
  send (Object1, Method_X, temp_variable => 'X');
  put_line("------------------------------------");
  put_line("Here is the Method_Y of the object1");
  send (Object1,Method_Y , temp_variable => 'Y');
  put_line("------------------------------------");
  put_line("Here is the Method_X of the object1");
  send (Object1, Method_X, temp_variable => 'Z');
  put_line("------------------------------------");
  put_line("Here is the Method_Y of the object1");
  send (Object1,Method_Y , temp_variable => 'W');
  put_line("------------------------------------");
  put_line("End of A1------------------------------------");
  new_line;

  put_line("Here is the beginning of the object2");
  Object2 := Alpha.Class_object;
  send (Object2, Create, new_instance => Object2);
  put_line("Here is the Method_X of the object2");
  send (Object2, Method_X, temp_variable => 'K');
  put_line("------------------------------------");
  put_line("Here is the Method_Y of the object2");
  send (Object2,Method_Y , temp_variable => 'L');
  put_line("------------------------------------");
  put_line("Here is the Method_X of the object2");
  send (Object2, Method_X, temp_variable => 'M');
  put_line("------------------------------------");
  put_line("Here is the Method_Y of the object2");
  send (Object2,Method_Y , temp_variable => 'N');
  put_line("------------------------------------");
  put_line("End of A2------------------------------------");
  new_line;

  put_line("Now we destroy the objects");

  send (Object1, Delete);

  send (Object2, Delete);

end program_mcv;
```

4. Program_mcv.script

```
Here is the beginning of the object1
in method create
Here is the Method_X of the object1
The value of X_mcv in this method is:

The new value of X_mv in this method is set and is:
X
-------------------------------------
Here is the Method_Y of the object1
The value of Y_mcv in this method is:

The new value of Y_mcv in this method is set and is:
Y
-------------------------------------
Here is the Method_X of the object1
The value of X_mcv in this method is:
X
The new value of X_mcv in this method is set and is:
Z
-------------------------------------
Here is the Method_Y of the object1
The value of Y_mcv in this method is:
Y
The new value of Y_mcv in this method is set and is:
W
-------------------------------------
End of A1----------------------------------

Here is the beginning of the object2
in method create
Here is the Method_X of the object2
The value of X_mcv in this method is:
Z
The new value of X_mcv in this method is set and is:
K
-------------------------------------
Here is the Method_Y of the object2
The value of Y_mcv in this method is:
W
The new value of Y_mcv in this method is set and is:
L
```

```
------------------------------------
Here is the Method_X of the object2
The value of X_mcv in this method is:
K
The new value of X_mcv in this method is set and is:
M
------------------------------------
Here is the Method_Y of the object2
The value of Y_mcv in this method is:
L
The new value of Y_mcv in this method is set and is:
N
------------------------------------
End of A2------------------------------------

Now we destroy the objects
```

B. CLASSES WITH INHERITANCE RELATION

1. Beta_spec_mcv.ca

```
class Beta is
  superclass Alpha;

   method Create (New_Instance : out Object_id );

  instance method Delete;

end Beta;
```

2. Beta_body_mcv.ca

```
with text_io;
use text_io;

Class body Beta is

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

 instance method Delete is
  begin
    send ( super, Delete);
    DESTROY;
  end Delete;

end Beta;
```

3. Program_mcv_inher.ca

```
with Alpha;
with Beta;
with text_io;
use text_io;

procedure program_mcv_inher is

 ObjectA1 : Object_id;
 ObjectA2 : Object_id;
 ObjectB1 : Object_id;
 ObjectB2 : Object_id;

begin

  put_line("Here is the beginning of the objectA1");
  ObjectA1 := Alpha.Class_object;
  send (ObjectA1, Create, new_instance => ObjectA1);
  put_line("Here is the Method_X of the ObjectA1");
  send (ObjectA1, Method_X, temp_variable => 'X');
  put_line("--------------------------------------");
  put_line("Here is the Method_Y of the ObjectA1");
  send (ObjectA1,Method_Y , temp_variable => 'Y');
  put_line("--------------------------------------");
  put_line("Here is the Method_X of the ObjectA1");
  send (ObjectA1, Method_X, temp_variable => 'Z');
  put_line("--------------------------------------");
  put_line("Here is the Method_Y of the ObjectA1");
  send (ObjectA1,Method_Y , temp_variable => 'W');
  put_line("--------------------------------------");
  put_line("End of A1--------------------------------------");
  new_line;

  put_line("Here is the beginning of the objectB1");
  ObjectB1 := Alpha.Class_object;
  send (ObjectB1, Create, new_instance => ObjectB1);
  put_line("Here is the Method_X of the ObjectB1");
  send (ObjectB1, Method_X, temp_variable => 'X');
  put_line("--------------------------------------");
  put_line("Here is the Method_Y of the ObjectB1");
  send (ObjectB1,Method_Y , temp_variable => 'Y');
  put_line("--------------------------------------");
put_line("Here is the Method_X of the ObjectB1");
send (ObjectB1, Method_X, temp_variable => 'Z');
put_line("------------------------------------");
put_line("Here is the Method_Y of the ObjectB1");
send (ObjectB1,Method_Y , temp_variable => 'W');
put_line("------------------------------------");
put_line("End of A1--------------------------------------");
new_line;

put_line("Here is the beginning of the ObjectA2");
ObjectA2 := Alpha.Class_object;
send (ObjectA2, Create, new_instance => ObjectA2);
put_line("Here is the Method_X of the ObjectA2");
send (ObjectA2, Method_X, temp_variable => 'K');
put_line("------------------------------------");
put_line("Here is the Method_Y of the ObjectA2");
send (ObjectA2,Method_Y , temp_variable => 'L');
put_line("------------------------------------");
put_line("Here is the Method_X of the ObjectA2");
send (ObjectA2, Method_X, temp_variable => 'M');
put_line("------------------------------------");
put_line("Here is the Method_Y of the ObjectA2");
send (ObjectA2,Method_Y , temp_variable => 'N');
put_line("------------------------------------");

put_line("Here is the beginning of the ObjectB2");
ObjectB2 := Alpha.Class_object;
send (ObjectB2, Create, new_instance => ObjectB2);
put_line("Here is the Method_X of the ObjectB2");
send (ObjectB2, Method_X, temp_variable => 'K');
put_line("------------------------------------");
put_line("Here is the Method_Y of the ObjectB2");
send (ObjectB2,Method_Y , temp_variable => 'L');
put_line("------------------------------------");
put_line("Here is the Method_X of the ObjectB2");
send (ObjectB2, Method_X, temp_variable => 'M');
put_line("------------------------------------");
put_line("Here is the Method_Y of the ObjectB2");
send (ObjectB2,Method_Y , temp_variable => 'N');
put_line("------------------------------------");
```

```
 put_line("Now we destroy the objects");
 send (ObjectA1, Delete);
 send (ObjectB1, Delete);
 send (ObjectA2, Delete);
 send (ObjectB2, Delete);
end program_mcv_inher;
```

4. Program_mcv_inher.script

```
Here is the beginning of the objectA1
in method create
Here is the Method_X of the objectA1
The value of X_mcv in this method is:

The new value of X_mcv in this method is set and is:
X
------------------------------------------
Here is the Method_Y of the objectA1
The value of Y_mcv in this method is:

The new value of Y_mcv in this method is set and is:
Y
------------------------------------------
Here is the Method_X of the objectA1
The value of X_mcv in this method is:
X
The new value of X_mcv in this method is set and is:
Z
------------------------------------------
Here is the Method_Y of the objectA1
The value of Y_mcv in this method is:
Y
The new value of Y_mcv in this method is set and is:
W
------------------------------------------

End of A1------------------------------------------

Here is the beginning of the objectB1
in method create
Here is the Method_X of the objectB1
The value of X_mcv in this method is:

The new value of X_mcv in this method is set and is:
X
------------------------------------------
Here is the Method_Y of the objectB1
The value of Y_mcv in this method is:
```

```
The new value of Y_mcv in this method is set and is:
Y
-----------------------------------
Here is the Method_X of the objectB1
The value of X_mcv in this method is:
X
The new value of X_mcv in this method is set and is:
Z
-----------------------------------
Here is the Method_Y of the objectB1
The value of Y_mcv in this method is:
Y
The new value of Y_mcv in this method is set and is:
W
-----------------------------------

End of B1-----------------------------------

Here is the beginning of the objectA2
in method create
Here is the Method_X of the objectA2
The value of X_mcv in this method is:

The new value of X_mcv in this method is set and is:
K
-----------------------------------
Here is the Method_Y of the objectA2
The value of Y_mcv in this method is:

The new value of X_mcv in this method is set and is:
L
-----------------------------------
Here is the Method_X of the objectA2
The value of X_mcv in this method is:
K
The new value of X_mcv in this method is set and is:
M
-----------------------------------
Here is the Method_Y of the objectA2
The value of Y_mcv in this method is:
L
```

```
The new value of X_mcv in this method is set and is:
N
-------------------------------------

End of A2-------------------------------------

Here is the beginning of the objectB2
in method create
Here is the Method_X of the objectB2
The value of X_mcv in this method is:

The new value of X_mcv in this method is set and is:
X
-------------------------------------
Here is the Method_Y of the objectB2
The value of Y_mcv in this method is:

The new value of Y_mcv in this method is set and is:
Y
-------------------------------------
Here is the Method_X of the objectB2
The value of X_mcv in this method is:
X
The new value of X_mcv in this method is set and is:
Z
-------------------------------------
Here is the Method_Y of the objectB2
The value of Y_mcv in this method is:
Y
The new value of Y_mcv in this method is set and is:
W
-------------------------------------

End of B2-------------------------------------

Now we destroy the objects
```

APPENDIX G - CONCURRENCY WITH METHOD VARIABLES

A. METHOD CLASS VARIABLE

1. Alpha_spec_CV.ca

```
class Alpha is

    method Create (New_Instance : out Object_id );

   instance method Method_X(temp_variable : in character);

   instance method Method_Y(temp_variable : in character);

   instance method Delete;

end Alpha;
```

2. Alpha_body_CV.ca

```
with text_io;
use text_io;

Class body Alpha is

 X_mcv : Character;
 Y_mcv : Character;

 method Create ( new_instance : out Object_id) is
  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

 instance method Method_X(temp_variable: in character) is
  begin
    put_line ("The value of X_mcv in this method is:");
    new_line;
    put(X_mcv);
    new_line;
    X_mcv := temp_variable;
    put_line ("The new value of mv in this method is set and is:");
    put(X_mcv);
    new_line;
  end Method_X;

 instance method Method_Y(temp_variable: in character) is
  begin
    put_line ("The value of Y_mcv in this method is:");
    new_line;
    put(Y_mcv);
    new_line;
    Y_mcv := temp_variable;
    put_line ("The new value of mv in this method is set and is:");
    put(Y_mcv);
    new_line;
  end Method_Y;
  instance method Delete is
   begin
     DESTROY;
   end Delete;

end Alpha;
```

3. Program_mcv_conc.ca

```
with Alpha;
with text_io; use text_io;

procedure program_mcv_conc is

pragma priority(1);

task t1 is
 pragma priority(1);
end;

task body t1 is
  Object1 : Object_id;
  begin
   put_line("Here is the beginning of the object1");
   Object1 := Alpha.Class_object;
   send (Object1, Create, new_instance => Object1);

    put_line("Here is the Method_X of the object1");
    send (Object1, Method_X, temp_variable => 'X');
    put_line("-----------------------------------");
     put_line("Here is the Method_Y of the object1");
    send (Object1,Method_Y , temp_variable => 'Y');
    put_line("-----------------------------------");
    put_line("Here is the Method_X of the object1");
    send (Object1, Method_X, temp_variable => 'Z');
    put_line("-----------------------------------");
     put_line("Here is the Method_Y of the object1");
    send (Object1,Method_Y , temp_variable => 'W');
    put_line("-----------------------------------");
    new_line;
    put_line("End of A1-------------------------------------");
    new_line;
end t1;

task t2 is
 pragma priority(1);
end;
```

```
task body t2 is
 Object2 : Object_id;

  begin

   put_line("Here is the beginning of the object2");
   Object2 := Alpha.Class_object;
   send (Object2, Create, new_instance => Object2);
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'K');
   put_line("------------------------------------");
   put_line("Here is the Method_Y of the object2");
   send (Object2,Method_Y , temp_variable => 'L');
   put_line("------------------------------------");
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'M');
   put_line("------------------------------------");
   put_line("Here is the Method_Y of the object2");
   send (Object2,Method_Y , temp_variable => 'N');
   put_line("------------------------------------");
   new_line;
   put_line("End of A2----------------------------------------");
   new_line;
end t2;

begin
   put_line("main");

 end program_mcv_conc;
```

4. Program_mcv_conc.script

```
Here is the beginning of the object2
in method create
Here is the Method_X of the object2
The value of X_mcv in this method is:


The new value of mv in this method is set and is:
K
-----------------------------------
Here is the Method_Y of the object2
The value of Y_mcv in this method is:


The new value of mv in this method is set and is:
L
-----------------------------------
Here is the Method_X of the object2
The value of X_mcv in this method is:

K
The new value of mv in this method is set and is:
M
-----------------------------------
Here is the Method_Y of the object2
The value of Y_mcv in this method is:

L
The new value of mv in this method is set and is:
N
-----------------------------------

End of A2-----------------------------------

Here is the beginning of the object1
in method create
Here is the Method_X of the object1
The value of X_mcv in this method is:

M
The new value of mv in this method is set and is:
X
```

```
------------------------------------
Here is the Method_Y of the object1
The value of Y_mcv in this method is:

N
The new value of mv in this method is set and is:
Y
------------------------------------
Here is the Method_X of the object1
The value of X_mcv in this method is:

X
The new value of mv in this method is set and is:
Z
------------------------------------
Here is the Method_Y of the object1
The value of Y_mcv in this method is:

Y
The new value of mv in this method is set and is:
W
------------------------------------

End of A1------------------------------------

main
```

B. METHOD INSTANCE VARIABLE

1. Alpha_spec_miv.ca

```
class Alpha is

    method Create (New_Instance : out Object_id );

    instance method Method_X(temp_variable : in character);

    instance method Method_Y(temp_variable : in character);

    instance method Delete;

end Alpha;
```

2. Alpha_body_miv.ca

```
with text_io;
use text_io;

Class body Alpha is

instance_variable1 : instance Character;
X_im :instance Character;
Y_im :instance Character;

method Create ( new_instance : out Object_id) is

  begin
    new_instance := INSTANTIATE ;
    put_line("in method create");
  end Create;

instance method Method_X(temp_variable: in character) is
  X_miv : Character;
  begin
    X_miv :=  X_im;
    put_line ("The value of X_miv in this method is:");
    new_line;
    put(X_miv);
    new_line;
    X_miv := temp_variable;
    put_line ("The new value of X_miv in this method is set and is:");
    put(X_miv);
    new_line;
    X_im := X_miv;
  end Method_X;

instance method Method_Y(temp_variable: in character) is
  Y_miv : Character;
  begin
    Y_miv :=  Y_im;
    put_line ("The value of Y_miv in this method is:");
    new_line;
    put(Y_miv);
    new_line;
    Y_miv := temp_variable;
    put_line ("The new value of Y_miv in this method is set and is:");
```

```
      put(Y_miv);
      new_line;
      Y_im :=Y_miv;
    end Method_Y;

  instance method Delete is
    begin
      DESTROY;
    end Delete;

end Alpha;
```

3. Program_miv_conc.ca

```
with Alpha;
with text_io; use text_io;

procedure program_miv_conc is
pragma priority(1);

task t1 is
 pragma priority(1);
end;

task body t1 is
 Object1 : Object_id;
 begin
   put_line("Here is the beginning of the object1");
   Object1 := Alpha.Class_object;
   send (Object1, Create, new_instance => Object1);
   put_line("Here is the Method_X of the object1");
   send (Object1, Method_X, temp_variable => 'X');
   put_line("--------------------------------------");
   put_line("Here is the Method_Y of the object1");
   send (Object1,Method_Y , temp_variable => 'Y');
   put_line("--------------------------------------");
   put_line("Here is the Method_X of the object1");
   send (Object1, Method_X, temp_variable => 'Z');
   put_line("--------------------------------------");
   put_line("Here is the Method_Y of the object1");
   send (Object1,Method_Y , temp_variable => 'W');
   put_line("--------------------------------------");
   new_line;
   put_line("End of A1--------------------------------------");
   new_line;
end t1;

task t2 is
 pragma priority(1);
end;
```

```
task body t2 is
 Object2 : Object_id;
 begin
   put_line("Here is the beginning of the object2");
   Object2 := Alpha.Class_object;
   send (Object2, Create, new_instance => Object2);
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'K');
   put_line("-----------------------------------");
   put_line("Here is the Method_Y of the object2");
   send (Object2,Method_Y , temp_variable => 'L');
   put_line("-----------------------------------");
   put_line("Here is the Method_X of the object2");
   send (Object2, Method_X, temp_variable => 'M');
   put_line("-----------------------------------");
   put_line("Here is the Method_Y of the object2");
   send (Object2,Method_Y , temp_variable => 'N');
   put_line("-----------------------------------");
   new_line;
   put_line("End of A2-----------------------------------");
   new_line;
end t2;

 begin
   put_line("main");

 end program_miv_conc;
```

4. Program_miv_conc.script

```
Here is the beginning of the object2
in method create
Here is the Method_X of the object2
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
K

-----------------------------------------
Here is the Method_Y of the object2
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
L

-----------------------------------------
Here is the Method_X of the object2
The value of X_miv in this method is:
K
The new value of X_miv in this method is set and is:
M

-----------------------------------------
Here is the Method_Y of the object2
The value of Y_miv in this method is:
L
The new value of Y_miv in this method is set and is:
N

-----------------------------------------
End of A2-----------------------------------------


Here is the beginning of the object1
in method create
Here is the Method_X of the object1
The value of X_miv in this method is:

The new value of X_miv in this method is set and is:
X
```

```
------------------------------------
Here is the Method_Y of the object1
The value of Y_miv in this method is:

The new value of Y_miv in this method is set and is:
Y

------------------------------------
Here is the Method_X of the object1
The value of X_miv in this method is:
X
The new value of X_miv in this method is set and is:
Z

------------------------------------
Here is the Method_Y of the object1
The value of Y_miv in this method is:
Y
The new value of Y_miv in this method is set and is:
W

------------------------------------
End of A1-----------------------------------

main
```

LIST OF REFERENCES

[AA90] Alsys Ada User Manuals 4.4 System, *Alsys Ada Compilation System for the Transputer*, Alsys Ada, May 1990.

[Act90] The Whitewater Group, *Actor User's Manual*, Volume 1 and Volume 2, The Whitewater Group, Inc., May 1990.

[Agh86] Agha, G., *Actors: A model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass, 1986.

[Bar89] Barnes, J.G.P., *Programming in Ada*(3rd Edition) , International Computer Science Series, Addison-Weley Publishers, ISBN 0-201-17566-5, 1989.

[BT88] Bal, H.E. and Tanenbaum, A.S., "Distributed Programming with Shared Data", *IEEE Int'l Conf. on Computer Language 1988*, pp. 82-91, Miami Beach FL, October 1988.

[Boo87] Brooch, G., *Software Engineering with Ada*(2nd Edition), Benjamin Cummings Publishing Company, Menlo Park CA 1987.

[Bro89] Bronnenberg, W., "POOL and DOOM", *Lecture Notes in Computer Science 365* , pp. 356-373, PARLE'89 vol I, Eindhoven, Netherlands, June 1989.

[BY87] Briot, J.P. and Yonezawa A., "Inheritance and Synchro-nization in Concurrent OOP", *Lecture Notes in Computer Science 276* , pp. 32-40, ECOOP'87, Paris, France 1987.

[BLW87] Burns, A., Lister, A.M., and Wellings, A.J., "A Review of Ada Tasking" *Lecture Notes in Computer Science 262* , Springer-Verlag Berlin Heidelberg 1987.

[BN91] Byrnes, R.B. and Nelson, M.L., "An Object-Oriented Simulation of Autonomous Underwater Vehicle" 22nd Annual Pittsburgh Conference on Modeling and Simulation, *Computers, Computer Architectures, Vision, Microprocessor in Education,* pp. 1581-1588, Pittsburgh, PA, May 1991.

[DW89] Dally, J.W. and Wills, D.S., "Universal Mechanisms for Concurrency", *Lecture Notes in Computer Science 365* , pp. 19-33, PARLE'89 vol I, Eindhoven, Netherlands, June 1989.

[DT88] Danforth, S. and Tominson, C., "Type Theories and Object- Oriented Programming" Vol 20, No 1 of *ACM Computing Surveys*, pp. 29-72, March 1988.

[dPN91] de Paula, E.G. and Nelson, M.L., "Designing a Class Hierarchy", *Proceedings of the Technology of Object-Oriented Languages and Systems International Conference 5 (TOOLS USA '91)*, pp. 203-218 Santa Barbara, CA, July 1991.

[HO87] Halbert, D.C. and O'Brien, P.D., "Using Types and Inheritance in Object-Oriented Programming" Vol 4, No 5 *IEEE Software*, pp. 71-79, September 1987.

[Has90] Hastings, A.B., "Distributed Lock Management in a Transaction Processing Environment" pp. 22-31, *IEEE Ninth Symposium on Reliable Distributed Systems*, Hunstsville, Alabama October 1990.

[Hor90] Horn C., "Is Object-Orientation a Good Thing for Distributed Systems?" *Lecture Notes in Computer Science 433* , Springer-Verlag Berlin Heidelberg 1990.

[INM89] INMOS, *The Transputer Handbook*, October 1989.

[Kim90] Kim, W., "Object-Oriented Databases: Definition and Research Directions" Vol 2, No 3 *IEEE Transactions on Knowledge and Data Engineering*, pp. 327-341, September 1990.

[KL89] Kim, W. and Lochovsky, F.H., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley Publishing, 1989.

[Low88] Low, C., "A Shared, Persistent Object Store", *Lecture Notes in Computer Science 322* , pp. 391-408, ECOOP'88, Oslo, Norway, August 1988.

[Mey88] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall International(UK), Hreartforshire, 1988.

[Mic88] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages" Vol 1, No 1, *JOOP* pp.12-34 April/May 1988.

[Nel90a] Naval Postgraduate School Report 52-90-024, *An Introduction to Object-Oriented Programming*, by Nelson, M.L., April 1990.

[Nel90b] Naval Postgraduate School Report 52-90-025, *Object-Oriented Database Management Systems*, by Nelson, M.L., May 1990.

[Nel90c] Naval Postgraduate School Report 52-90-026, *Concurrent Object-Oriented Systems*, by Nelson, M.L., September 1990.

[Nel91a] Nelson, M.L., "An Object-Oriented Tower of Babbel", Vol 2, No 3, *OOPS Messanger* pp. 3-11 July 1991.

[Nel91b] Nelson, M.L., "Concurrency & Object-Oriented Programming" *SIGPLAN* pp. 63-72, Vol 26, No 10, October 1991.

[Ng90] Ng, T.N., "The Design and Implementation of a Reliable Distributed Operating System-ROSE" pp. 2-11, *IEEE Ninth Symposium on Reliable Distributed Systems,* Hunstsville, Alabama October 1990.

[NMO90] Nelson, M.L., Moshell, J.M., and Orooji, A., *"A Relational Object-Oriented Management Systems"* pp. 319-323 International Phoenix Conference on Computers and Communications (IPCCC'90), March 1990.

[NM92] Nelson, M.L. and Mota, R., "Object-Oriented Programming in Classic-Ada" (draft).

[NS88] Nielsen, K. and Shumate, K., *Designing Large Real-Time Systems with Ada,* McGraw-Hill, New York, 1988

[OM88] Oudshoorn, M. and Marlin, C., "Describing Data Control in Programming Languages", *IEEE Int'l Conf. on Computer Language 1988,* pp. 100-109, Miami Beach Florida, October 1988.

[RB91] Rumbaugh, J., Blaha, M., and others, *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Seb89] Sebesta, W.S., *Concepts of Programming Languages,* The Benjamin/Cummings Publishing Company, Redwood City, CA 1989.

[SM88] Shaer, S. and Mellor, S.J., *Object-Oriented Systems Analysis: Modeling the world in data,* Englewood Cliffs, NJ, 1988.

[SW87] Shriver, B. and Wegner, P., *Research Directions in Object-Oriented Programming,* MIT Press Series in Computer Systems, Caimbridge, Mass., 1987.

[Sor88] Sorgaard, P., "Object-Oriented Programming and Computerized Shared Material", *Lecture Notes in Computer Science 322* , pp. 319-334, ECOOP'88, Oslo, Norway, August 1988.

[SB86] Stefik, M. and Bobrow, D.G., "Object-Oriented Programming: Themes and Variations", pp. 40-62 *The AI Magazine* Winter 1986.

[SN90] Steigerwald,R.A. and Nelson, M.L., "Concurrent Programming in Smalltalk-80" Vol 25, No 28 of *SIGPLAN Notices* pp.27-39 August 1990.

[Sof89] Software Productivity Solutions, *Classic-Ada User's Manual*, Software Productivity Solutions, Indialantic, FL, 1989.

[TS90] Tully, A. and Shrivastava, S.K., "Preventing State Divergence in Replicated Distributed Programs" pp. 104-113, *IEEE Ninth Symposium on Reliable Distributed Systems*, Hunstsville, Alabama October 1990.

[Weg87] Wegner, P., "Dimensions of Object-Based Language Design" *OOPSLA'87* pp. 168-182, October 1987.

[WP88] Wiener, R.S. and Pinson, L.J., *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley Publishing Co, Reading, Mass, 1988.

[WWW90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3. MAJ Michael L. Nelson, USAF Computer Science Department, Code CSNe Naval Postgraduate School Monterey, CA 93943	2
4. CDR Gary J. Hughes, USN Computer Science Department, Code CSHu Naval Postgraduate School Monterey, CA 93943	2
5. Embassy of Greece Naval Attache 2228 Massachusettes Ave.,N.W. Washington, D.C. 20008	4
5. LT Vassilios Theologitis, Hellenic Navy G. Blessa 11 Papagou 15669 Athens, GREECE	3